Nina Moebius

Modellgetriebene Entwicklung sicherer Smart Card-Anwendungen

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
an der Fakultät für Angewandte Informatik
der Universität Augsburg, 2012

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Tag der mündlichen Prüfung:	05.03.2013

Zusammenfassung

Chipkartenanwendungen gewinnen weltweit zunehmend an Bedeutung. Immer mehr Menschen nutzen elektronische Geldbörsen und es gibt kaum noch eine Metropole, die keinen elektronischen Fahrausweis für ihren Nahverkehr anbietet. Viele Firmen setzen bei ihren Schließsystemen auf Transponderlösungen als Zugangskontrolle und in Deutschland steht aktuell die Einführung der elektronischen Gesundheitskarte an. All diesen Chipkartenanwendungen ist gemein, dass sowohl das Ausnutzen von Sicherheitslücken als auch deren Behebung erhebliche finanzielle Schäden bei den Betreibern verursachen und deren Ruf nachhaltig beschädigen können. Gleichzeitig gilt für viele Anwendungen, dass die Akzeptanz durch die Nutzer ganz entscheidend dadurch beeinflusst wird, wie sicher ihre teils hochsensiblen, persönlichen Daten in den Systemen sind.

Die Praxis zeigt, dass Sicherheit in diesem Umfeld ein großes Problem darstellt. Immer wieder werden Sicherheitslücken in Anwendungen entdeckt, die zu dem Zeitpunkt bereits jahrelang im Einsatz waren. Die Behebung dieser Fehler gestaltet sich aufgrund der vielen Chipkarten, die an die Nutzer einer Anwendung ausgegeben wurden, sowie der relativ hohen Zahl an Kartenlesegeräten oft schwierig und ist sehr kostspielig.

Was bei der Entwicklung dieser Art von Anwendungen bisher fehlt ist ein Softwareentwicklungsansatz, der die Sicherheit der Anwendung als integralen Bestandteil begreift und diesen Aspekt durchgehend in den Ansatz einbettet. Die Entwickler von Chipkartenanwendungen setzen teilweise bereits formale Methoden ein, um Sicherheitseigenschaften nachzuweisen. Dennoch fehlt bislang ein Konzept, das die Sicherheitsanforderungen der zu entwickelnden Systeme vom Entwurf bis zur Implementierung durchgängig berücksichtigt und mit dem gleichzeitig Garantien für die Sicherheit gegeben werden können.

Diese Arbeit stellt einen modellgetriebenen Softwareentwicklungsansatz vor, in dem die beiden oben genannten Aspekte umgesetzt werden. Der Ansatz umfasst den Entwurf der zu entwickelnden Anwendung mit UML und die vollautomatische Generierung von lauffähigem Quellcode aus dem UML-Anwendungsmodell. Einzigartig ist, dass neben dem eigentlichen Quellcode auch ein formales, abstraktes Modell der Anwendung generiert wird. Dieses kann automatisch in ein interaktives Verifikationswerkzeug eingelesen und es kann dann ein formaler Beweis der Sicherheit der Anwendung geführt werden. Auf diese Weise wird garantiert, dass die Anwendung kritische Eigenschaften erfüllt. Der Quellcode und das formale Modell werden so generiert, dass sich die auf formaler Ebene bewiesenen Sicherheitseigenschaften auf den generierten Code übertragen. Die Eigenschaften gelten somit auch für den Quellcode der Anwendung. Der gesamte Ansatz ist werkzeugunterstützt und wurde anhand mehrerer, teils sehr großer, Fallstudien evaluiert.

Danksagung

Diese Arbeit wäre nicht möglich gewesen, wenn nicht viele Menschen zu ihrem Gelingen beigetragen hätten. Mein Dank gilt im Besonderen:

meinem Doktorvater **Prof. Dr. Wolfgang Reif** für seine Unterstützung und sein Vertrauen in mich, seine Motivation und Anmerkungen sowie für all die Dinge, die ich in den Jahren hier am Lehrstuhl gelernt habe.

Dr. Kurt Stenzel für die vielen konstruktiven Diskussionen, die wir zusammen geführt haben, seine Hilfe bei der Implementierung des SecureMDD-Frameworks, seine unendliche Geduld sowie seine Fähigkeit, Dinge immer wieder zu hinterfragen.

Kuzman Katkalov für seine Arbeit als wissenschaftliche Hilfskraft und die damit verbundene Implementierung von Teilen der Modelltransformationen, die im Rahmen seiner Diplomarbeit entstandenen Ergebnisse zum Thema „modellgetriebenes Testen“ sowie für das Korrekturlesen dieser Seiten.

Marian Borek für seine Arbeit als wissenschaftliche Hilfskraft, die ebenfalls die Implementierung der Transformationen vorangebracht hat.

Dr. Holger Grandy für die ersten, aber dennoch nicht weniger wichtigen Diskussionen zu der Modellierung und Codegenerierung, seine Ergebnisse zur Verfeinerung von Java-Anwendungen sowie seine Unterstützung in meiner Anfangszeit am Lehrstuhl.

Dr. Dominik Haneberg für seine Forschungsergebnisse zum Thema „Verifikation von Smart Card-Anwendungen“, auf denen diese Arbeit aufbaut, sowie das aufmerksame Korrekturlesen von großen Teilen der Arbeit.

Dr. Florian Nafz für das intensive Korrekturlesen dieser Dissertation und die vielen kleinen Tipps und Hinweise, die mir das Schreiben erleichtert haben.

Ebenfalls danke ich **Dr. Gerhard Schellhorn** für die Korrekturen und Anmerkungen zum formalen Teil der Arbeit.

Ein ganz besonderer Dank gilt **meinem Mann Jonathan** und **meinem Sohn Linus** für ihre Unterstützung, ihre Geduld und ihr Verständnis in den letzten Monaten und Jahren. Ebenfalls danken möchte ich meiner weiteren Familie, insbesondere **meiner Mutter** und **meinem Bruder Philipp**, die immer hinter mir stehen und mich ebenfalls sehr unterstützt haben.

Inhaltsverzeichnis

Teil I.

Der SecureMDD-Ansatz: Motivation und Überblick

1 Einleitung

1.1. Motivation

Die Anzahl der Chipkarten, die jeder von uns im Portemonnaie bei sich hat, nimmt immer mehr zu. Die Verwendung der kleinen Plastikkarten mit Chip hat sich als sehr praktisch erwiesen und setzt sich im täglichen Leben mehr und mehr durch. Smart Cards sind Mini-Computer, die einen Mikroprozessor, einen Speicher sowie einen Eingabe- und einen Ausgabekanal besitzen. Ein großer Vorteil von Chipkarten ist ihr manipulationsresistenter Speicher, auf dem geheime Daten sicher gespeichert werden können.

Die Anwendungen, bei denen Smart Cards zum Einsatz kommen, sind vielfältig. So enthalten nicht nur der deutsche Personalausweis sowie der elektronische Reisepass mittlerweile eine Chipkarte, auch die elektronische Gesundheitskarte, die zurzeit an die gesetzlich Versicherten ausgegeben wird und die bisherige Versichertenkarte ersetzt, ist eine Smart Card. Außerdem besitzt ein Großteil der Bevölkerung eine EC-Karte, die inzwischen in der Regel neben einem Magnetstreifen auch einen Chip besitzt. Viele Universitäten verwenden eine Universitätskarte, die verschiedene auf dem Campus benötigte Anwendungen vereint. Dies sind neben dem Zahlungsverkehr in der Mensa auch der Bibliotheksausweis und z.B. eine Kopierkartenanwendung zum Bezahlen von Kopien. Viele Firmen verwenden Zugangskontrollsysteme, die ebenfalls auf Chipkarten basieren. Zahlreiche Systeme, zum Beispiel elektronische Ticketsysteme für den Personennahverkehr oder Stadienkarten in Sportarenen und Veranstaltungszentren runden dieses Bild ab.

Es ist daher nicht verwunderlich, dass die Zahl der ausgegebenen Smart Cards von Jahr zu Jahr ansteigt. *Eurosmart*, eine Gesellschaft, die die Smart Card Security Industrie vertritt [14], veröffentlicht jährlich die Zahlen der ausgelieferten Smart Cards (inklusive der SIM-Karten in Mobiltelefonen). Im Jahr 2011 wurden weltweit über 6,1 Milliarden Karten ausgegeben, für das Jahr 2012 wird ein Anstieg um ca. 13 Prozent auf ca. 6,9 Milliarden Karten erwartet. Die Prognose ist, dass sich dieser Trend in den kommenden Jahren fortsetzt.

Für den Einsatz von Smart Cards benötigt man neben den Chipkarten noch Geräte, die mit den Chipkarten kommunizieren können. Diese Geräte werden in der Chipkartenwelt als Terminals bezeichnet. Für die Kommunikation mit einer Karte wird ein Kartenlesegerät verwendet, das an ein Terminal angeschlossen ist. Dabei unterscheidet man zwischen kontaktbehafteten und kontaktlosen Karten. Die Kommunikation der Terminals mit den Chipkarten ist durch

kryptographische Protokolle festgelegt. Die Protokolle sorgen dafür, dass die Kommunikation nach festen Regeln erfolgt und kryptographisch gegen Angriffe abgesichert ist, zum Beispiel durch Verschlüsselung der übertragenen Daten. Die Sicherheit einer Chipkartenanwendung beruht darauf, dass beim Entwurf der kryptographischen Protokolle, die in der Regel abhängig von der entwickelten Anwendung sind, keine Fehler gemacht werden. Das Problem hierbei ist, dass der Entwurf eines kryptographischen Protokolls sehr schwierig ist und selbst kleine Protokolle mit wenigen Kommunikationsschritten sehr anfällig für Fehler und Sicherheitslücken sind [2,5,171]. Das bekannte Zitat des Informatikers und Kryptographie-Experten Roger Needham, „Security protocols are three line programs that people still manage to get wrong“ [42], trifft diese Beobachtung sehr gut.

Weiß man von dieser Fehleranfälligkeit, ist man nicht überrascht, dass in den Medien immer wieder von Angriffen auf Systeme berichtet wird, die auf Unsicherheiten in den zugrunde liegenden Protokollen zurückzuführen sind. Ein bekanntes Beispiel im Bereich Smart Card-Anwendungen ist der von Ross Anderson und Kollegen im Jahr 2010 entdeckte Angriff auf das EMV-Verfahren [141], das für Zahlungen mit EC- und Kreditkarten verwendet wird. Gegenüber heise Security [76] äußerte der Sicherheitsexperte Anderson, dass in dem EMV-Verfahren vermutlich weitere Schwachstellen zu finden sind. Er ist sich sicher, dass mindestens eine davon auch betrügerische und bisher ungeklärte Abhebungen an Bankautomaten erkläre, die innerhalb der letzten zwei Jahre gehäuft aufgetreten sind [75].

Durch die Ausnutzung von Sicherheitslücken entstehen jedes Jahr enorme Kosten und wirtschaftliche Schäden für die betroffenen Firmen und Banken. Möglicherweise genauso bedeutend wie die materiellen Schäden ist der drohende Verlust der Privatsphäre durch solche Angriffe, denn der Trend geht immer mehr dahin, dass Chipkartenanwendungen sensible und persönliche Daten von Privatpersonen speichern und verarbeiten. Beispiele hierfür sind der elektronische Reisepass (digitales Foto, Fingerabdruck, Name, Anschrift, Geburtsdatum) oder die elektronische Gesundheitskarte (Informationen über chronische Krankheiten und Patientenakten). Dadurch ist auch der Schutz von Daten vor dem Zugriff unberechtigter Personen ein essentieller Teil des Entwurfs einer Smart Card-Anwendung. Dies geht einher mit dem Recht einer Person auf informationelle Selbstbestimmung. Diese besagt, dass eine Person selbst über die Weitergabe und Verwendung ihrer personenbezogenen Daten entscheiden darf. Gelangt eine unbefugte Person durch eine Sicherheitslücke an personenbezogene Daten, ist dies nicht nur ein Verstoß gegen das Bundesdatenschutzgesetz, sondern führt möglicherweise auch zu immensen Image- und Akzeptanzproblemen bei den Nutzern der Anwendung.

Dies führt zu der Frage, wie ein Entwicklungsprozess für sichere Chipkartenanwendungen auszusehen hat. Beim Entwurf und der Realisierung einer Anwendung reicht es nicht aus, sich nur auf die Funktionalität der Anwendung zu konzentrieren. Vielmehr muss die Betrachtung der Sicherheit der Anwendung in den kompletten Entwicklungsprozess integriert und auf diese Weise ein Software-Security Co-Design möglich sein. Um für die entwickelte Anwendung Sicherheitseigenschaften garantieren zu können, ist die Integration von formaler Verifikation in den Ansatz essentiell.

In dieser Arbeit wird ein modellgetriebener Entwicklungsansatz für Smart Card-Anwendungen vorgestellt, der diese Kriterien erfüllt. Mit dem Ansatz ist es möglich, eine Anwendung vollständig zu modellieren und aus diesen Modellen automatisch lauffähigen Quellcode zu generieren. Um die Sicherheit der Anwendung garantieren zu können, wird aus dem Modell außerdem automatisch eine formale Spezifikation generiert, die für die interaktive Verifika-

tion von Sicherheitseigenschaften verwendet wird. Auf diese Weise können Garantien für die entwickelte Anwendung gegeben werden.

1.2. Überblick über die Arbeit

In Abbildung 1.1 sind der Aufbau der Arbeit sowie die Abhängigkeiten zwischen den einzelnen Kapiteln grafisch dargestellt. Die Arbeit lässt sich in fünf Teile untergliedern.

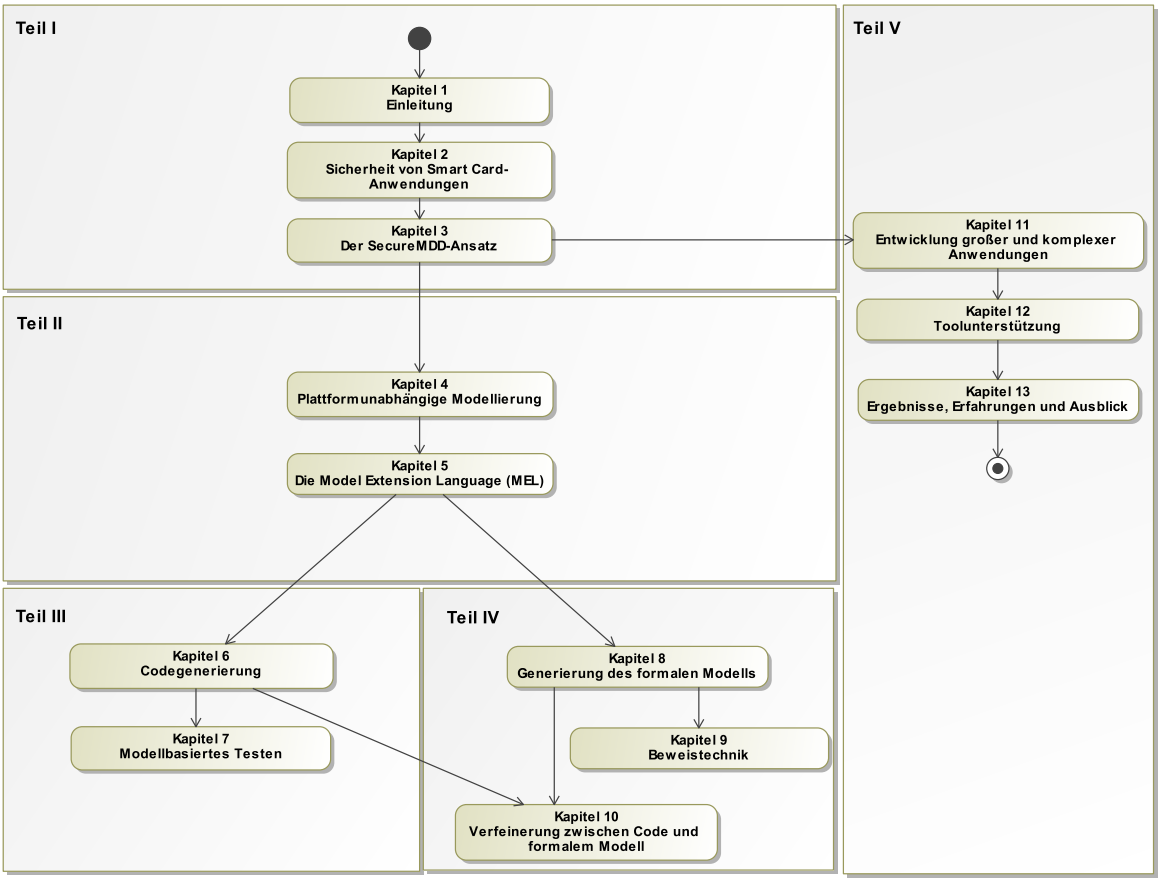

Abbildung 1.1.: Überblick über den Aufbau der Arbeit

Der erste Teil gibt einen Überblick über den in dieser Arbeit vorgestellten Entwicklungsansatz. Kapitel 2 veranschaulicht anhand von einigen Beispielen aus der Praxis, weshalb die Entwicklung von Anwendungen, die auf kryptographischen Protokollen basieren, schwierig und fehleranfällig ist. Im Anschluss daran wird der Begriff „Sicherheit“ im Kontext dieser Arbeit diskutiert.

In Kapitel 3 wird der modellgetriebene Ansatz für die Entwicklung sicherer Chipkartenanwendungen vorgestellt und in das DFG-Projekt SecureMDD eingeordnet, in dessen Rahmen die Arbeit entstand. Außerdem erläutert das Kapitel die in dieser Arbeit referenzierten Fallstudien und stellt den Ansatz in den Kontext anderer Forschungsarbeiten zu diesem Thema.

Im zweiten Teil wird die in dieser Arbeit entwickelte Modellierung einer Chipkartenanwendung mit UML [73] beschrieben. Kapitel 4 erläutert die plattformunabhängige Modellierung mit UML. Hierfür werden zunächst die definierten Sicherheitsdatentypen und das UML-Profil vorgestellt. Im Anschluss wird detailliert auf die einzuhaltenden Modellierungsrichtlinien für die verschiedenen Diagrammtypen eingegangen. Weiterhin wird die Modellierung anhand einer Fallstudie illustriert.

In Kapitel 5 ist die Model Extension Language (MEL) beschrieben. Dies ist eine domänenspezifische Sprache, die im Rahmen dieser Arbeit für die Modellierung der dynamischen Aspekte einer sicherheitskritischen Anwendung entworfen wurde. Das Kapitel erläutert wichtige Entscheidungen, die beim Entwurf der Sprache getroffen wurden. Außerdem beschrieben sind die abstrakte Syntax und Semantik, die konkrete Syntax sowie eine Auflistung der in MEL definierten Operationen, zum Beispiel zum Verschlüsseln eines Datums oder das Erstellen eines Zertifikats.

Der dritte Teil erläutert die Generierung des Quellcodes aus dem UML-Modell sowie das Testen des Codes. Kapitel 6 beschreibt zunächst allgemein die Übersetzung der einzelnen UML-Diagramme in Code und geht dann speziell auf die Generierung des Codes für die Terminals und die Chipkarten ein.

In Kapitel 7 wird das modellbasierte Testen einer mit dem vorgestellten Ansatz entwickelten Anwendung erläutert. Das Kapitel gliedert sich auf in die Modellierung von Testfällen mit UML und die Generierung der Testfälle aus den Modellen. Die Modellierung der Testfälle umfasst dabei die Initialisierung der zu testenden Chipkarten und Terminals sowie die Modellierung von konkreten funktionalen Tests und Angriffen.

Der vierte Teil beschäftigt sich mit den formalen Aspekten des Ansatzes. Da für die betrachteten Anwendungen die Sicherheit eine zentrale Rolle spielt, ist die Integration von formalen Methoden essentiell. Nur durch ihren Einsatz können garantierte Aussagen über die Sicherheit einer Anwendung getroffen werden. Kapitel 8 beschreibt die automatische Generierung einer formalen Spezifikation aus dem erstellten UML-Modell. Diese ist eine vollständige abstrakte Beschreibung der Anwendung und spezifiziert außerdem einen Angreifer, der versucht, das System zu manipulieren und dabei (wirtschaftliche) Schäden anzurichten.

Kapitel 9 erläutert die für die Verifikation verwendete Beweistechnik basierend auf dem aus algebraischen Spezifikationen und Abstract State Machines bestehenden formalen Modell.

In Kapitel 10 wird die Verfeinerungsbeziehung zwischen dem generierten Code und formalen Modell diskutiert. Bei der Implementierung der Transformationen wurde speziell darauf geachtet, dass diese Beziehung gilt, denn sie ist die Basis dafür, dass die auf dem formalen Modell bewiesenen Sicherheitseigenschaften auch für den generierten Code gelten.

Im fünften Teil wird auf die Entwicklung großer und umfangreicher Anwendungen mit dem vorgestellten Ansatz eingegangen und die Toolunterstützung für den Ansatz erläutert. Kapitel 11 stellt ein inkrementelles Vorgehen vor, um die Komplexität großer Anwendungen zu beherrschen. Die Entwicklung großer Anwendungen ist am Beispiel der elektronischen Gesundheitskarte illustriert. Diese ist von ihrem Umfang, von ihrer Komplexität und aufgrund

der vielen beteiligten Komponenten und Benutzer sehr gut dafür geeignet. Anschließend wird anhand einiger Zahlen verdeutlicht, welchen Umfang die betrachteten Fallstudien haben und wie viel Aufwand für die Verifikation nötig war.

Kapitel 12 beschreibt die für die Entwicklung mit dem SecureMDD-Ansatz zur Verfügung gestellte Werkzeugkette. Diese umfasst ein Tool zur Validierung des UML-Modells sowie das Einlesen, Validieren und Verarbeiten der im UML-Modell enthaltenen MEL-Ausdrücke. Weiterhin sind die implementierten Modelltransformationen durch Eclipse-Plugins realisiert, die ein einfaches Ausführen der Transformationen auf einem Modell ermöglichen.

In Kapitel 13 sind die in dieser Arbeit erzielten Ergebnisse zusammengefasst. Das Kapitel gibt außerdem einen Ausblick auf mögliche Erweiterungen.

Die verwandten Forschungsarbeiten sind in den entsprechenden Kapiteln der Dissertation erläutert.

Hinweis: Die Artefakte der durchgeführten Fallstudien sowie die implementierten Modelltransformationen sind zu umfangreich, um sie vollständig in dieser Arbeit anzugeben. Die UML-Modelle der Fallstudien, der automatisch generierte Code sowie die formalen Spezifikationen sind deshalb auf der Webseite des SecureMDD-Projekts[1] zu finden. Dort einzusehen sind auch die in KIV durchgeführten Beweise für einige Fallstudien. Der Quellcode der Modelltransformationen ist in [129] und [128] veröffentlicht.

1.3. Erreichte Ergebnisse

Die dieser Dissertation zugrunde liegende Forschungsarbeit hat einen modellgetriebenen, integrierten Softwareentwicklungsansatz für sichere Chipkartenanwendungen hervorgebracht. Unseres Wissens nach ist dies die erste Arbeit, die einen vollständigen Entwicklungsansatz für sicherheitskritische Smart Card-Anwendungen definiert. Vollständig bedeutet in diesem Kontext sowohl ein Software-Security Co-Design von Beginn der Entwicklung an als auch die Integration formaler Methoden sowie die gleichzeitige vollautomatische Generierung lauffähigen Codes. Die einzelnen Ergebnisse dieser Arbeit sind im Folgenden kurz zusammengefasst:

- **Modellierungsrahmen für sicherheitskritische Anwendungen**
  Es wurde ein Modellierungsrahmen definiert, mit dem sich sicherheitskritische Smart Card-Anwendungen mit UML modellieren lassen. Um eine intuitive Modellierung möglich zu machen, wurde UML durch Definition eines Profils auf die Domäne der sicherheitskritischen Anwendungen zugeschnitten. Außerdem wurde für die Erweiterung von UML-Aktivitätsdiagrammen eine externe domänenspezifische Sprache, genannt MEL, entwickelt, die für die Modellierung von kryptographischen Protokollen verwendet werden kann. Für ein UML-Modell ist außerdem eine Validierungskomponente implementiert, die prüft, ob die vorgegebenen Modellierungsrichtlinien eingehalten werden.
- **Automatische Generierung von lauffähigem Quellcode**
  Aus dem mit UML erstellten Modell der Anwendung wird mittels Modelltransformationen lauffähiger Quellcode generiert. Eine ergänzende Implementierung des Codes von Hand ist nicht notwendig. Generiert wird Code sowohl für Smart Cards als auch für Terminals, bei denen es sich um Automaten oder PCs mit integriertem Chipkartenlese-

[1] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/

gerät handelt. Insbesondere bei der Generierung des Smart Card-Codes werden von den Transformationsregeln z.B. die Ressourcenbeschränkung einer Chipkarte sowie deren eingeschränktes Kommunikationsverhalten besonders berücksichtigt.

- **Modellbasiertes Testen einer Anwendung**
  Die Arbeit stellt einen Ansatz zur Verfügung, mit dem der generierte Code getestet werden kann. Das Testen dient dem schnellen Finden von Fehlern und Sicherheitslücken. Unterstützt werden sowohl funktionale Tests als auch das Testen auf mögliche Angriffe. Die Spezifikation eines Testfalls erfolgt ebenfalls mit UML. Aus dem Testfall-Modell wird automatisch Code generiert, mit dem der zuvor generierte Code getestet werden kann.
- **Automatische Generierung einer formalen Spezifikation**
  Smart Card-Anwendungen sind sehr anfällig für Fehler im Design der zugrunde liegenden Protokolle. Gleichzeitig ist die Sicherheit einer solchen Anwendung essentiell. Aus dem UML-Modell wird deshalb automatisch mittels Modelltransformationen eine formale Spezifikation generiert, mit der die Sicherheit der Anwendung nachgewiesen werden kann. Diese verwendet algebraische Spezifikationen für die Formalisierung der Daten, der Kommunikationsstruktur und des Angreifers sowie Abstract State Machines [25] für die Spezifikation der kryptographischen Protokolle. Die generierte Spezifikation basiert auf der Arbeit von Haneberg [78], wurde jedoch an die Modellierung mit UML angepasst und signifikant verbessert.
- **Formale Verifikation der Sicherheit**
  Es wird eine Beweistechnik präsentiert, mit der sich die zuvor definierten Sicherheitseigenschaften der Anwendung verifizieren lassen. Dabei werden anwendungsspezifische Sicherheitseigenschaften betrachtet, die in den allermeisten Fällen bessere Garantien für die betrachteten E-Commerce-Anwendungen geben können, als Standardsicherheitseigenschaften. Die Grundlage für die verwendete Beweistechnik wurde durch die Arbeit von Haneberg [78] geschaffen.
- **Verfeinerung zwischen generiertem Code und dem formalen Modell**
  Damit die auf abstrakter Ebene bewiesenen Sicherheitseigenschaften auch auf den Code übertragen werden können, muss der generierte Code korrekt sein und darf keine zusätzlichen Sicherheitslücken enthalten. Dies ist sichergestellt, wenn der Code eine Verfeinerung der generierten formalen Spezifikation ist (siehe [66]). Diese Verfeinerungsbeziehung wurde bei der Implementierung der Transformationen für den Code und das formale Modell berücksichtigt, d.h. es kann angenommen werden, dass der Code eine Verfeinerung des formalen Modells ist. Ein formaler Nachweis der Verfeinerungsbeziehung und der dafür benötigte Korrektheitsbeweis der Transformationen ist nicht Teil dieser Arbeit.
- **Unterstützung der Entwicklung großer Anwendungen**
  Es wird anhand des Beispiels der elektronischen Gesundheitskarte gezeigt, dass die Entwicklung einer komplexen und großen Anwendung mit dem vorgestellten Ansatz ebenfalls möglich ist. Hierfür wird ein inkrementelles Vorgehen bei der Entwicklung vorgeschlagen und Handreichungen für die Umsetzung gegeben.
- **Toolunterstützung des Ansatzes**
  Der gesamte Entwicklungsansatz ist werkzeugunterstützt. Das mit einem UML-Model-

lierungstool erstellte Modell lässt sich mit wenigen Benutzerinteraktionen in Code und ein formales Modell transformieren. Das Laden der generierten Spezifikation in das interaktive Verifikationssystem KIV [9] erfolgt ebenfalls automatisch. Alle Modell-zu-Modell- und Modell-zu-Text-Transformationen sind vollständig implementiert. Eine vor den Transformationen stattfindende Validierung prüft das Eingabemodell automatisch auf Einhaltung der Modellierungsrichtlinien und korrekte Syntax. Der Ansatz wurde anhand von vielen Fallstudien evaluiert und konsolidiert und wird auch in der Lehre eingesetzt.

2 Sicherheit von Smart Card-Anwendungen

Zusammenfassung: Der Entwurf von Anwendungen, die auf kryptographischen Protokollen basieren, ist überraschend schwierig und selbst kurze Protokolle sind sehr anfällig für Fehler [5]. Dadurch können Sicherheitslücken in der Anwendung entstehen, die zum Teil erst nach mehreren Jahren und nach vielen ausgegebenen Smart Cards gefunden werden. Dies hat zur Folge, dass eine Korrektur des Fehlers und ein damit verbundenes Deployment des Softwareupdates auf alle betroffenen Chipkarten und Terminals oftmals sehr kostspielig ist. Dieses Kapitel zeigt anhand von mehreren Beispielen aus der Praxis, wie leicht Fehler in kommerziellen Anwendungen übersehen werden und welche Folgen die Verwendung solcher Anwendungen haben kann. Dies motiviert das Vorgehen nach einem (modellgetriebenen) Ansatz, der durch die integrierte Verifikation Sicherheitseigenschaften für die entwickelte Anwendung garantiert. Außerdem wird in diesem Kapitel der Begriff „Sicherheit“ diskutiert.

Abschnitt 2.1 illustriert anhand von fünf Beispielen wie fehleranfällig kryptographische Protokolle sind und welche Auswirkungen diese Fehler in der Praxis haben können. Abschnitt 2.2 erläutert den Begriff „Sicherheit“ und welche Arten von Sicherheitseigenschaften durch den in dieser Arbeit vorgestellten Entwicklungsansatz sichergestellt werden können.

2.1. Beispiele für Anwendungen mit Sicherheitslücken

2.1.1. Das EMV-Verfahren: „Chip and PIN is Broken“

Das EMV-Verfahren ist eine Spezifikation für die Zahlung mit EC- und Kreditkarten, die mit einem Prozessorchip ausgestattet sind. EMV setzt sich aus den Anfangsbuchstaben der drei Gesellschaften zusammen, die den EMV-Standard entwickelt haben (Eurocard International, MasterCard und Visa). Das EMV-Verfahren ersetzt das zuvor verwendete Verfahren für Karten, die nur mit einem Magnetstreifen und keinem Chip ausgestattet sind, denn Magnetstreifen gelten aufgrund der möglichen Skimming-Angriffe mittlerweile als unsicher. EMV ist

das am weitesten verbreitete Verfahren für Zahlungen mit Smart Cards in Europa. Im Jahre 2010 waren ca. 730 Millionen EMV-Karten im Umlauf.

Im Jahr 2010 entdeckten britische Forscher einen „Man-in-the-Middle“-Angriff auf das EMV-Protokoll [141]. Mit diesem Angriff ist es möglich, die EC- bzw. Kreditkarte davon zu überzeugen, dass das Terminal keine PIN-Eingabe, sondern lediglich eine Unterschrift verlangt. Dem Terminal wird dagegen vorgetäuscht, dass die Karte eine zuvor eingegebene PIN akzeptiert. Die eingegebene PIN kann dabei beliebig gewählt werden. Dieser Angriff ist möglich, weil die Bestätigung der Karte, dass die richtige PIN eingegeben wurde, nicht kryptographisch abgesichert ist.

Immer wieder kam es in der Vergangenheit zu Betrugsfällen, bei denen mit gestohlenen EC- und Kreditkarten in Geschäften und Tankstellen gezahlt wurde, obwohl für diese Zahlungen eigentlich eine PIN hätte eingegeben werden müssen. Ob diese Betrugsfälle durch Ausnutzen der oben genannten Sicherheitslücke zustande kamen ist möglich, konnte jedoch nicht geklärt werden.

2.1.2. Mifare Classic

Mifare Classic wurde 1995 entwickelt und war 2009 mit 70% Marktanteil die am häufigsten verwendete kontaktlose Smart Card weltweit. 2009 waren über eine Milliarde dieser Karten im Einsatz. Mifare Classic wird in vielen Städten als Bezahlsystem im öffentlichen Personennahverkehr verwendet. Beispiele sind die Oyster Card in London, die Luas Card in Dublin, die Charlie Card in Boston oder die SUBE Card in Buenos Aires. Weiterhin wird Mifare Classic von vielen Firmen für die Zugangskontrolle und von Universitäten, zum Beispiel der Universität Augsburg, als Mensakarte verwendet. Hersteller von Mifare Classic ist NXP[1], eine Ausgründung von Philips Semiconductors.

2008 veröffentlichten Forscher der Radboud University Nijmegen, basierend auf der Arbeit von Courtois et al. [43], erste Angriffe auf Mifare Classic [60]. Nach und nach wurden immer mehr Fehler entdeckt, die in der Summe katastrophale Folgen haben. Die Hauptursachen für die Angriffe waren verschiedene Protokollfehler, ein deterministischer Zufallszahlengenerator sowie ein unsicherer Kryptoalgorithmus [61]. Ein Protokollfehler besteht darin, dass eine Mifare Classic-Karte in einem Authentisierungsschritt als Nachweis ihrer Echtheit die ihr zugeschickten Daten verschlüsselt, dabei allerdings die Herkunft der Daten nicht überprüft. Auf diese Weise ist ein Known-Plaintext-Angriff möglich. Dadurch werden Teile des Verschlüsselungsstroms und damit indirekt auch Teile des geheimen Schlüssels offengelegt. Ein weiterer Protokollfehler ist die mehrfache Verwendung eines One-Time-Pads.

Als Konsequenz gibt es auf die Mifare Classic-Karte mindestens vier verschiedene Angriffe, die die geheimen Schlüssel der Karten errechnen können. Damit ist es möglich, sich als legitimiertes Terminal auszugeben, Mifare Classic-Karten zu klonen und damit Dienstleistungen auf Kosten des Kartenbesitzers zu nutzen. Auch ein kostenloses Aufladen der Prepaid-Karten ist möglich. Da diese Karten kontaktlose RFID-Systeme sind, ist dies auch im Vorbeigehen (mit bis zu zehn Metern Distanz) möglich, wenn eine Person eine Mifare Classic-Karte bei sich trägt. Als Lösung hat NXP die Mifare Plus-Karte entwickelt. Es gibt aber auch heute noch viele Anwendungen, die Mifare Classic-Karten verwenden. Der Grund hierfür ist, dass

[1] www.nxp.com

der Austausch aller im Umlauf befindlichen Karten sowie aller Terminals sehr kostspielig ist. Die Zahl der Betrugsfälle sowie die dadurch entstandenen Kosten wurden jedoch nicht veröffentlicht.

2.1.3. Needham-Schroeder, SMB und TLS Renegotiation

Dieser Abschnitt gibt einen Überblick über drei weitere fehlerhafte Protokolle.

Ein klassisches Beispiel, das im Zusammenhang mit Designfehlern in kryptographischen Protokollen sehr oft genannt wird, ist das Needham-Schroeder-Protokoll [143]. Das Protokoll ermöglicht die Authentisierung zweier Kommunikationspartner und realisiert außerdem den anschließenden Austausch eines Sitzungsschlüssels. Es besteht aus sieben Kommunikationsschritten. Das Protokoll wurde nicht speziell für Smart Card-Anwendungen entworfen, verdeutlicht aber sehr gut, dass auch kleine Protokolle mit nur wenigen Kommunikationsschritten Sicherheitslücken enthalten können. Das Needham-Schroeder-Protokoll wurde 1978 entwickelt, 1981 zeigten Denning und Sacco einen Replay-Angriff (d.h. das Wiedereinspielen von Nachrichten) in dem Protokoll auf [47] und korrigierten diesen. 1995 entdeckte Lowe mittels Model-Checking einen neuen Angriff auf das korrigierte Protokoll und beseitigte die Schwächen im Jahr darauf [117]. Das Needham-Schroeder Protokoll ist ein theoretisches, in der Literatur sehr häufig zitiertes Beispiel, das jedoch keinen praktischen Einsatz fand. Eine Weiterentwicklung dieses Protokolls ist das Netzwerkauthentisierungsprotokoll Kerberos [144], das auch heute noch verwendet wird.

Das Server Message Block-Protokoll (SMB) [125] ist ein Netzwerkprotokoll, das die gemeinsame Verwendung von Dateien für Druck-, Datei- und andere Serverdienste realisiert. Das Protokoll wurde 1983 von IBM entwickelt und im Laufe der Jahre von verschiedenen Firmen, insbesondere von der Firma Microsoft, weiterentwickelt. SMB wird u.a. von den Microsoft Betriebssystemen verwendet. Im Jahr 2001 wurde ein Angriff auf das SMB-Protokoll bekannt, der SMB Reflection-Angriff [118]. Ein Reflection-Angriff ist eine Möglichkeit, um ein (Challenge-Response) Authentisierungssystem anzugreifen, das dasselbe Verfahren sowohl für die Authentisierung des Clients als auch des Servers verwendet. In diesem Beispiel muss der Angreifer Zugriff auf den Server haben. Nach einem erfolgreichen Angriff hat der Server die Möglichkeit, sich auf dem Client als angemeldeter Benutzer auszugeben. Ein Angreifer, der einen SMB Reflection-Angriff erfolgreich ausgeführt hat, kann die vollständige Kontrolle über das betroffene System übernehmen. Von dem Angriff waren zahlreiche Windows Betriebssysteme betroffen, die Lücke wurde 2008 durch einen Patch geschlossen. Der durch den Angriff entstandene Schaden ist nicht bekannt.

Das Transport Layer Security-Protokoll (TLS, [50]) ist die Weiterentwicklung des SSL-Protokolls. Es realisiert den Austausch eines Sitzungsschlüssels zwischen Client und Server und beinhaltet außerdem die zertifikatbasierte Authentisierung beider Seiten. Mit dem TLS-Protokoll ist es möglich, die Eigenschaften der Verbindung innerhalb einer Sitzung neu zu verhandeln (renegotiation). Dies ermöglicht beispielsweise, dass sich zunächst nur der Server authentisiert und die Authentisierung des Clients erst zu einem späteren Zeitpunkt erfolgt. Auch können zum Beispiel verwendete Schlüssellängen zu einem späteren Zeitpunkt neu bestimmt werden. Das TLS-Protokoll wurde 1999 entworfen und wird u.a. von HTTPS, IMAP und LDAP verwendet. Im November 2009 wurde ein Fehler im TLS-Protokoll bekannt, der sich auf die Neuverhandlung von Sitzungsparametern bezieht. Mit dieser „Man-in-the-Middle“-Attacke ist es

möglich, die Neuverhandlung zu manipulieren und Daten in die Kommunikation zwischen Client und Server einzuschleusen, von denen der Empfänger annimmt, dass sie vom authorisierten Kommunikationspartner gesendet wurden. Reale Angriffe, die diesen Protokollfehler ausgenutzt haben, sind nicht bekannt. Jedoch zeigt ein Angriff auf die Blogging-Plattform Twitter, der zu Demonstrationszwecken gemacht wurde, dass der Fehler auch praktische Relevanz hat. Die Lücke wurde im Januar 2010 durch eine Überarbeitung des Protokolls geschlossen.

2.2. Diskussion des Begriffs „Sicherheit“

Eine Anwendung ist sicher, wenn sie keine Schwachstellen oder Sicherheitslücken besitzt. Diese Formulierung ist jedoch recht allgemein gehalten und deshalb nur schwer für eine konkrete Anwendung zu überprüfen. Aus diesem Grund ist es beim Entwurf einer Anwendung notwendig, sich Gedanken über die Sicherheitsziele und möglichen Angriffspunkte und Bedrohungen des zu entwickelnden Systems zu machen und Sicherheitseigenschaften, die für die Anwendung gelten müssen, zu erfassen und zu formulieren. Da man nicht sicherstellen kann, dass dabei an alle möglichen Bedrohungen gedacht wurde, hat es sich in der Praxis durchgesetzt, lieber etwas mehr „Sicherheit“ in Form von z.B. Verschlüsselung von Daten, Nachrichtenintegrität oder den Schutz vor Replay-Angriffen in die kryptographischen Protokolle zu integrieren, als nötig scheint.

Weiterhin haben verschiedene Benutzergruppen und auch der Betreiber einer E-Commerce-Anwendung unterschiedliche Interessen bezüglich der Sicherheit der Anwendung. Der Betreiber möchte zum Beispiel, dass ihm durch einen Angriff kein Geld verloren geht. Dies bedeutet, dass der Angreifer (der auch ein Benutzer der Anwendung sein kann) nicht in der Lage ist, sich einen finanziellen Vorteil durch den Angriff zu verschaffen. Den Benutzern dagegen ist es zum Beispiel wichtig, dass ihre persönlichen Daten vor einem Angreifer geschützt sind und sie eine Anwendung trotz Anwesenheit und Manipulation des Angreifers störungsfrei nutzen können. Bei der Erfassung der Sicherheitseigenschaften müssen somit die Aspekte aus Sicht der einzelnen Benutzergruppen separat voneinander analysiert werden.

Außerdem spielt bei der Entwicklung von Softwaresystemen die Wirtschaftlichkeit eine Rolle. Hierfür muss anhand einer Kosten-Nutzen-Analyse berechnet werden, ob die Mehrkosten, die bei der Entwicklung einer sichereren, aber dafür komplexeren Anwendung entstehen, die entstehenden Kosten bei Auftreten eines Angriffs übersteigen. Insbesondere bei Smart Card-Anwendungen, bei denen eine große Zahl an Smart Cards ausgegeben und verwendet wird, ist die Frage von Bedeutung, ob ein Angriff nur einzelne Karten unsicher macht (zum Beispiel durch Erraten des privaten Schlüssels dieser Karten) oder alle ausgegebenen Karten betroffen sind.

Sicherheitseigenschaften, die von einer konkreten Anwendung abhängen, werden in dieser Arbeit als anwendungsspezifische Sicherheitseigenschaften bezeichnet. Diese sind von Anwendung zu Anwendung verschieden und stehen oft in direktem Zusammenhang mit der durch die Anwendung bereitgestellten Funktionalität. Eine solche anwendungsspezifische Eigenschaft ist zum Beispiel, dass beim Übertragen von Geld von einer Geldkarte auf eine andere Karte auch bei einem Angriff kein Geld verloren gehen kann. Dies bedeutet, dass der Geldbetrag, der von einer Karte abgebucht wurde, auch sicher auf die andere Karte aufgebucht wird. Eine weitere anwendungsspezifische Sicherheitseigenschaft ist, dass ein elektronisches Rezept, das auf der elektronischen Gesundheitskarte gespeichert ist, nur einmal in einer Apotheke

eingelöst werden kann. Damit wird verhindert, dass sich ein Patient die doppelte Menge des verschriebenen Medikaments ausgeben lässt. Die kryptographischen Protokolle, die anwendungsspezifische Sicherheitseigenschaften sicherstellen sollen, sind in der Regel ebenfalls abhängig von der Anwendung.

Neben den anwendungsspezifischen Eigenschaften gibt es noch anwendungsunabhängige Sicherheitseigenschaften, die in dieser Arbeit auch als Standardeigenschaften bezeichnet werden. Dies sind zum Beispiel die Geheimhaltung oder Integrität von Daten oder die Authentisierung einzelner Kommunikationspartner. Für diese Eigenschaften gibt es in der Regel Standardtechniken und -protokolle, die in verschiedenen Kontexten, d.h. für verschiedene Anwendungen, verwendet werden können. Einige Beispiele für Standardprotokolle werden in [5, 28, 117] vorgestellt.

Mit dem in dieser Arbeit vorgestellten modellgetriebenen Ansatz werden in der Regel anwendungsspezifische Sicherheitseigenschaften nachgewiesen. Sie geben in den meisten Fällen bessere Garantien für eine Anwendung als Standardeigenschaften. Die Verifikation von Standardeigenschaften ist mit dem Ansatz jedoch ebenfalls möglich. Standardeigenschaften sind sogar meistens die Voraussetzung dafür, dass eine anwendungsspezifische Eigenschaft gilt, d.h. diese Eigenschaften müssen dann ebenfalls bewiesen werden.

Ob eine Anwendung sicher ist oder nicht hängt auch davon ab, welche Fähigkeiten ein Angreifer für seine Angriffe besitzt. In dieser Arbeit kann für jede Kommunikationsverbindung spezifiziert werden, ob ein Angreifer Nachrichten, die über die Verbindung geschickt werden, mitlesen oder unterdrücken kann. Außerdem kann angegeben werden, ob er eigene Nachrichten verschicken kann. Dies ist ein Unterschied zum Dolev Yao-Angreifermodell [51], bei dem der Angreifer vollen Zugriff (Lesen, Senden, Unterdrücken) auf alle Kommunikationskanäle hat. Es wird außerdem davon ausgegangen, dass der Angreifer ebenfalls ein legaler Benutzer der Anwendung sein kann und so z.B. seine eigene PIN-Nummer kennt. Weiterhin wird angenommen, dass der Angreifer keinen direkten Zugriff auf die Terminals hat, d.h. die Installation von Schadsoftware auf einem Terminal, um sich Zugang zu dem Gerät zu verschaffen und das anschließende Auslesen der dort gespeicherten Daten wird nicht betrachtet. Smart Cards sind manipulationssichere Speicher und somit gut geeignet für die Speicherung von geheimen Schlüsseln und Daten, die ein Angreifer nicht erfahren darf. Dass ein Angreifer Zugriff auf den Speicher einer Smart Card hat, kann nicht hunderprozentig ausgeschlossen werden. Der Aufwand hierfür, zum Beispiel durch Differential Power Analysis [109], ist jedoch extrem hoch [159], weshalb davon ausgegangen wird, dass der Speicher einer Chipkarte manipulationssicher ist und nicht ausgelesen werden kann. Zusammenfassend kann man sagen, dass der Angreifer eine „Black Box“-Sicht hat, in der er die versendeten Nachrichten mitlesen und manipulieren kann, aber keinen direkten Zugriff auf die Terminals und Smart Cards hat.

3 Der modellgetriebene Softwareentwicklungsansatz

Zusammenfassung: Der modellgetriebene Ansatz zur Entwicklung sicherer Chipkartenanwendungen basiert auf dem Ansatz der Model Driven Architecture (MDA) [69]. Er verwendet UML inklusive einiger Erweiterungen für die abstrakte Modellierung einer Anwendung und erstellt alle weiteren Artefakte, bis hin zum Quellcode, automatisch. Eine Besonderheit ist die Integration der formalen Verifikation, um die Sicherheit der entworfenen Anwendung zu garantieren sowie die integrierte Möglichkeit des modellbasierten Testens der entworfenen Anwendung, um auf einfache Weise Fehler in den zugrunde liegenden Protokollen zu finden. In diesem Kapitel wird zunächst eine Einführung in die modellgetriebene Softwareentwicklung gegeben. Anschließend werden der SecureMDD-Entwicklungsansatz sowie die durchgeführten Fallstudien vorgestellt und verwandte Arbeiten erläutert. Der Inhalt dieses Kapitels ist in [133] publiziert.

Abschnitt 3.1 erläutert die in dieser Arbeit benötigten Grundlagen der modellgetriebenen Entwicklung, der UML-Profile und der domänenspezifischen Sprachen. Abschnitt 3.2 stellt den modellgetriebenen Entwicklungsansatz für Smart Card-Anwendungen vor. Abschnitt 3.3 beschreibt die durchgeführten Fallstudien und Abschnitt 3.4 erläutert verwandte Arbeiten zur modellgetriebenen Entwicklung sicherheitskritischer Anwendungen.

3.1. Grundlagen zur modellgetriebenen Softwareentwicklung

3.1.1. Die Model Driven Architecture

Der SecureMDD-Ansatz folgt dem Ansatz der modellgetriebenen Softwareentwicklung (Model Driven Software Development, MDSD) [179]. In dessen Mittelpunkt steht das Modell. Dieses wird nicht nur, wie in vielen Softwareentwicklungsansätzen üblich, zur Dokumentation verwendet, sondern auch, um mithilfe von Modelltransformationen (semi-)automatisch andere Artefakte zu generieren. In der Regel sind dies weitere Modelle und Quellcode. Um in der Lage zu sein, weitere Artefakte aus den Ursprungsmodellen zu generieren, müssen die verwendeten Modelle eine hohe Ausdruckskraft haben. Insbesondere ist eine starke Formalisierung

der Modelle sowie eine klare Definition der Semantik notwendig. Ein möglicher und häufig verwendeter Weg, um modellgetriebene Softwareentwicklung in der Praxis zu realisieren, ist die Model Driven Architecture (MDA) [69, 155]. Diese ist ein Standard, der von der Object Management Group (OMG) [150] entwickelt wurde.

Die Grundidee der Model Driven Architecture ist es, Business- und Anwendungslogik von den darunter liegenden Technologien zu separieren. Hierdurch kommt es zu einer Trennung der fachlichen von den technischen Aspekten einer Anwendung. Für die fachliche Modellierung werden oft domänenspezifische Sprachen (DSL) [57] verwendet, die in der Regel speziell auf einen Anwendungsbereich zugeschnitten sind. Beide Punkte erleichtern einem Entwickler den Entwurf einer Anwendung. Die OMG legt nicht fest, in welcher Modellierungssprache die Modelle erstellt werden sollen. In der Praxis hat sich hierfür aber die Unified Modeling Language (UML) [73] durchgesetzt. Diese ist der de-facto Standard in der Modellierung von (Teilen einer) Software und ist durch die Definition von UML-Profilen leicht erweiterbar sowie an einen Problembereich anpassbar.

MDA verwendet den recht allgemein gehaltenen Begriff der „Plattform“. Eine Plattform kann z.B. ein Betriebssystem oder eine bestimmte Programmiersprache sein. Im Kontext dieser Arbeit werden sowohl die Terminals als auch die Smart Cards als eigene Plattformen betrachtet.

Die Model Driven Architecture definiert verschiedene Ebenen beginnend mit einer abstrakten Sicht, die dann Schritt für Schritt in immer konkretere Modelle für die verschiedenen Zielplattformen verfeinert wird. Dabei werden vier Ebenen bzw. Sichten definiert, die jeweils eine andere Perspektive auf eine Anwendung geben. Die von der MDA unterschiedenen Ebenen sind:

- das **Computation Independent Model (CIM)** beschreibt die Konzepte einer Anwendung und ihre Beziehungen zueinander. Es ist vergleichbar mit einem Konzept- oder Domänenmodell. Aus dem CIM werden keine anderen Modelle oder Code generiert.
- das **Platform Independent Model (PIM)** definiert die Systemfunktionalität. Es beschränkt sich auf die fachlichen Aspekte und abstrahiert von den technischen Details der Implementierung und der verwendeten Implementierungssprache.
- das **Platform Specific Model (PSM)** enthält mehr technische Details als das PIM. Aus einem PIM können mehrere PSMs generiert werden, da jede Zielplattform durch ein eigenes PSM repräsentiert werden kann.
- die **Platform Specific Implementation (PSI)** beschreibt die Implementierung, die aus einem PSM generiert wird. Da es für eine Anwendung mehrere PSMs geben kann, kann es auch mehrere PSIs geben.

Die MDA definiert außerdem zwei verschiedene Arten von Transformationen: Modell-zu-Modell-Transformationen (M2M) sowie Modell-zu-Text-Transformationen (M2T). Modell-zu-Modell-Transformationen transformieren ein Modell in ein anderes. Das Ausgangsmodell wird dabei als Quellmodell, das erzeugte Modell als Zielmodell bezeichnet. Zu jedem verwendeten Modell gibt es ein Metamodell. Dieses legt die verwendbaren Modellierungselemente sowie ihre Beziehungen zueinander fest. Das UML-Metamodell beispielsweise definiert, dass eine Klasse Attribute und Operationen besitzen kann. Ein Modell ist somit eine Instanz des zugehörigen Metamodells. Eine Modell-zu-Modell-Transformation bildet die Elemente des Quell-Metamodells auf die Elemente des Ziel-Metamodells ab. Quell- und Ziel-Metamodell können

dabei unterschiedlich sein. Im SecureMDD-Ansatz wird UML für alle Modelle verwendet, d.h. in allen Transformationen wird das UML-Metamodell verwendet. Lässt man eine Transformation auf einem Quellmodell laufen, transformiert diese die konkreten Elemente des Modells in Elemente des Zielmodells. Dieser Zusammenhang ist in Abbildung 3.1 veranschaulicht.

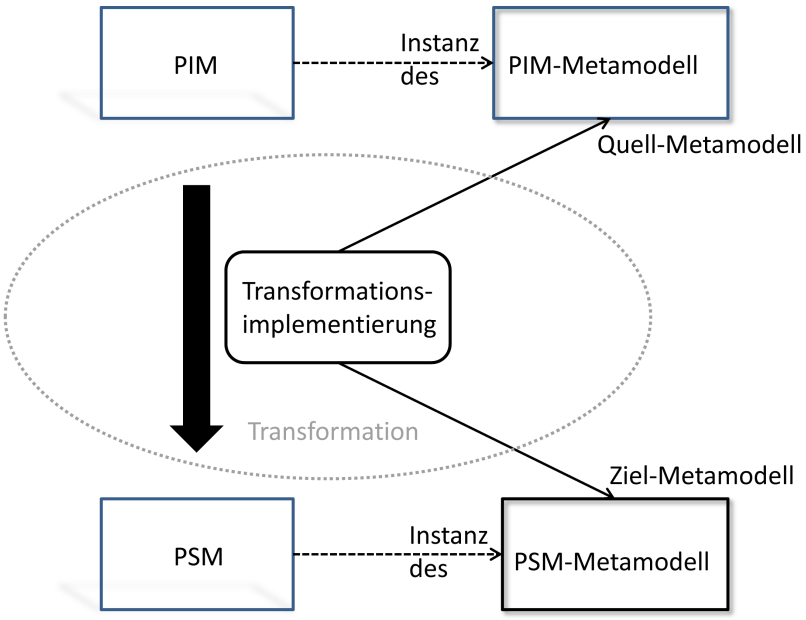

Abbildung 3.1.: Grafische Darstellung einer Modell-zu-Modell-Transformation

Eine Modell-zu-Text-Transformation übersetzt ein Modell in Text (Strings). Dies kann beliebiger Text sein, oft werden M2T-Transformationen für die Generierung von Quellcode verwendet. Die Transformationen sind, wie bei einer M2M-Transformation, relativ zum Quell-Metamodell beschrieben.

Die MDA legt die Anzahl der Transformationsschritte nicht fest. Ein PIM wird immer in ein PSM transformiert. Das so erzeugte PSM kann dann in Text, die sogenannte plattformspezifische Implementierung, transformiert werden. Es ist jedoch auch möglich, erneut eine Modell-zu-Modell-Transformation anzuwenden, die ein weiteres PSM erzeugt. In diesem Fall wird das Quellmodell (das in der vorherigen Transformation erzeugt wurde) wieder als PIM betrachtet. Auf diese Weise ist eine beliebige Abfolge von M2M-Transformationen möglich. Teilt man komplexe Modell-zu-Modell-Transformationen auf mehrere Transformationen auf, bleiben diese übersichtlicher und es wird außerdem eine schrittweise Konkretisierung der Modelle erreicht. Eine plattformspezifische Implementierung kann jedoch nicht weiter transformiert werden, sondern bildet das Endprodukt.

Die Vorteile der modellgetriebenen Softwareentwicklung liegen auf der Hand: Hat man die benötigten domänenspezifischen Sprachen und Erweiterungen sowie die Modelltransformationen erstellt, kostet das Modellieren einer Anwendung sowie die Generierung aller weiteren Artefakte deutlich weniger Zeit als das Erstellen von Hand. Außerdem ist die automatische

Generierung deutlich weniger fehleranfällig als das manuelle Erstellen der Artefakte. Ist die fachliche und technische Trennung durchgehend umgesetzt, so erlaubt die abstrakte, plattformunabhängige Modellierung einer Anwendung eine leichte Erweiterbarkeit. Hierfür können zusätzliche Plattformen definiert werden. Zusätzlich kann sich der Modellierer beim Entwurf auf die fachlichen Aspekte der Anwendung konzentrieren, während die technischen Details außen vor bleiben.

3.1.2. UML-Profile

Die Unified Modeling Language kann durch die Definition eines UML-Profils erweitert werden. Profile werden oft verwendet, um die UML an eine Anwendungsdomäne anzupassen. Ein UML-Profil ist eine leichtgewichtige Erweiterung des UML-Metamodells. Eine Besonderheit ist, dass Profile das UML-Metamodell (M2-Level des OMG-Schichtenmodells) selber nicht verändern, sondern auf der Ebene der Modelle (M1-Level) erstellt werden. Im Gegensatz zur schwergewichtigen Erweiterung des UML-Metamodells hat dies den Vorteil, dass bestehende UML-Modellierungstools verwendet werden können. Mit diesen Tools können sowohl eigene UML-Profile erstellt, als auch bestehende Profile zur Modellierung importiert und verwendet werden. Ein UML-Profil besteht aus `Stereotypen`, `Tagged Values` (Eigenschaftswerte) und `Constraints` (Einschränkungen), die bestehende UML-Metamodellelemente erweitern.

3.1.3. Domänenspezifische Sprachen (DSLs)

Eine domänenspezifische Sprache wird ebenfalls direkt für ein Anwendungsgebiet entworfen und ist auf dieses zugeschnitten. Das Ziel einer DSL ist es, alle Probleme der Anwendungsdomäne ausdrücken zu können. Im Gegensatz zu einer DSL stehen allgemein verwendbare Sprachen wie beispielsweise Java oder C.

Man unterscheidet zwischen internen und externen DSLs. Eine interne DSL ist in eine Sprache eingebettet und verwendet einige Elemente der Wirtssprache. Beispiele für interne DSLs sind UML-Profile oder das XUnit-Framework. Im Gegensatz dazu sind externe DSLs eigenständige Sprachen, die eine eigene Syntax und Semantik besitzen. Sie sind in der Verwendung somit flexibler als interne DSLs, der Aufwand für die Erstellung ist jedoch höher.

Der SecureMDD-Ansatz definiert sowohl ein UML-Profil (siehe Abschnitt 4.1.3) als auch eine externe domänenspezifische Sprache (siehe Kapitel 5) für die Modellierung von Anwendungen, die auf kryptographischen Protokollen basieren.

3.2. Überblick über den SecureMDD-Entwicklungsansatz

Der SecureMDD-Ansatz ermöglicht die modellgetriebene Softwareentwicklung sicherheitskritischer Smart Card-Anwendungen, die auf kryptographischen Protokollen basieren. Meiner Meinung nach kann eine Anwendung nur dann sicher sein, wenn die Sicherheitsaspekte schon von Beginn der Entwicklung an berücksichtigt werden. Aus diesem Grund ist die Betrachtung der Sicherheit vollständig in den Entwicklungsansatz integriert. Ihre garantierte Einhaltung ist dabei durch die Verwendung formaler Verifikation sichergestellt. Dieses Software-Security Co-Design ermöglicht auf einfache Weise den Entwurf einer sicherheitkritischen Anwendung

und garantiert gleichzeitig deren Sicherheit. Der Fokus des Ansatzes liegt dabei auf verteilten Smart Card-Anwendungen. Diese haben gemeinsam, dass sie auf kryptographischen Protokollen basieren, die von Natur aus schwierig zu entwerfen sind. Beim Entwurf von SecureMDD wurde Wert darauf gelegt, dass der Ansatz erweiterbar ist. Dabei bedeutet Erweiterbarkeit die Unterstützung weiterer Anwendungsgebiete, zum Beispiel die Anbindung von Services oder mobilen Anwendungen, an denen Smartphones beteiligt sind. Erweiterbarkeit bedeutet aber auch die Unterstützung weiterer Verifikationstools, zum Beispiel den Anschluss eines Model Checkers, sowie die Unterstützung weiterer kryptographischer Operationen. Der bestehende Ansatz ist auch für die Entwicklung großer und komplexer Anwendungen gut geeignet. Dies wurde anhand der Fallstudie „elektronische Gesundheitskarte“ evaluiert.

Ein Überblick über den SecureMDD-Entwicklungsansatz ist in Abbildung 3.2 dargestellt.

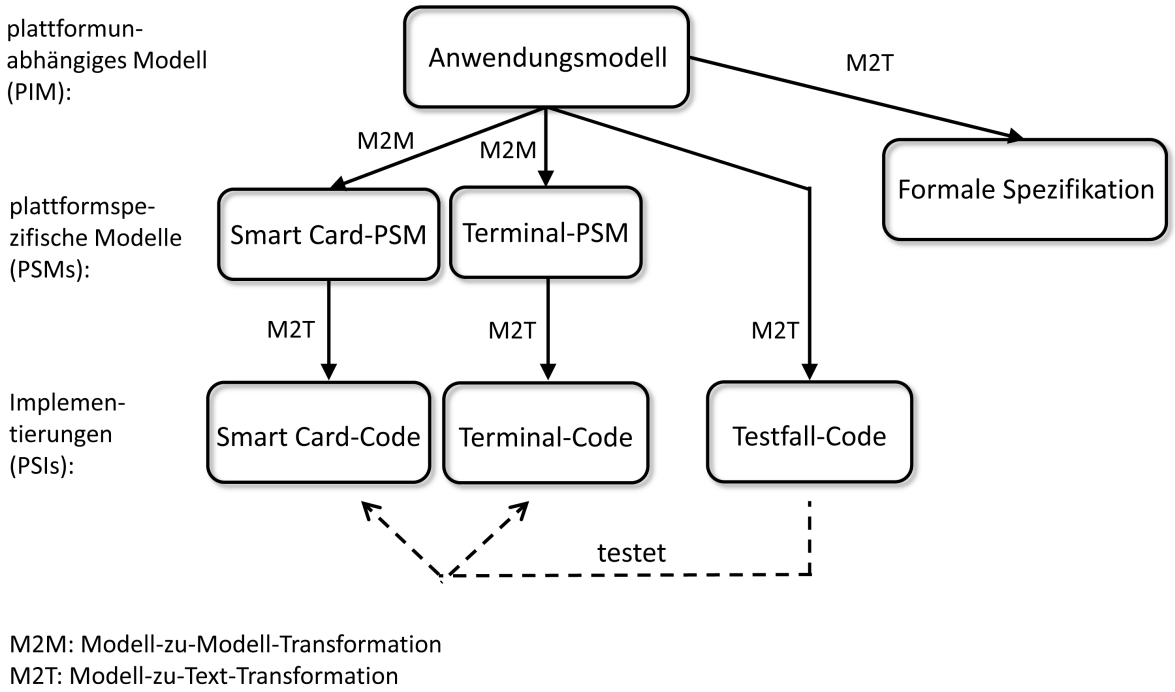

Abbildung 3.2.: Überblick über den SecureMDD-Entwicklungsansatz

Der erste Schritt bei der Entwicklung einer Anwendung ist das Erstellen eines plattformunabhängigen UML-Modells (PIM). Dieses Anwendungsmodell abstrahiert von den technischen Details der Anwendung. Die im plattformunabhängigen Modell dargestellte abstrakte Sicht einer Anwendung ermöglicht es dem Entwickler, sich auf die Funktionalität und die der Anwendung zugrunde liegenden kryptographischen Protokolle sowie die Sicherheit der Anwendung zu konzentrieren. Das plattformunabhängige Modell enthält sowohl die statischen als auch die dynamischen Aspekte einer Anwendung. Das Modell wird in UML erstellt. Für die Modellierung der statischen Aspekte sowie des Angreifers werden Klassen- und Deploymentdiagramme verwendet. Die dynamischen Teile, d.h. die kryptographischen Protokolle, sind mit Sequenz- und Aktivitätsdiagrammen modelliert. Um die UML an die Domäne der sicherheitskritischen Anwendungen anzupassen, wurde sie um ein UML-Profil erweitert. Zusätzlich wurde eine

domänenspezifische Sprache (DSL) definiert, die die UML-Aktivitätsdiagramme ergänzt und es ermöglicht, die dynamische Sicht vollständig mit Aktivitätsdiagrammen zu modellieren. Die DSL ist objekt-orientiert und ermöglicht u.a. das Erzeugen von Objekten, Zuweisungen, Deklaration von lokalen Variablen und das Durchführen kryptographischer und arithmetischer Operationen auf den modellierten Datentypen. Das plattformunabhängige Modell ist zwar eine abstrakte Sicht einer Anwendung, enthält jedoch alle Details, die für die Generierung des Quellcodes der Anwendung sowie das Erstellen einer formalen Spezifikation zum Beweis der Sicherheit benötigt werden. Dies bedeutet, dass die aus dem plattformunabhängigen Modell erzeugten plattformspezifischen Modelle, der Code sowie die generierte formale Spezifikation nicht von Hand ergänzt oder geändert werden müssen. Ein plattformunabhängiges Modell kann werkzeugunterstützt validiert werden. Bei der Validierung wird überprüft, ob das Anwendungsmodell gültig ist, d.h. die Modellierungsrichtlinien eingehalten werden und keine Syntaxfehler, zum Beispiel bei der Verwendung der DSL, mehr vorhanden sind. Die plattformunabhängige Modellierung einer Anwendung sowie das definierte UML-Profil sind in Kapitel 4 beschrieben. Die domänenspezifische Sprache wird in Kapitel 5 erläutert.

Aus dem plattformunabhängigen Anwendungsmodell werden plattformspezifische UML-Modelle (PSMs) generiert. Für jeden Komponententyp, d.h. Smart Cards und Terminals, wird ein plattformspezifisches Modell erzeugt. Die Erstellung der plattformspezifischen Modelle ist ein Zwischenschritt, der vor der Generierung des Quellcodes stattfindet. Ziel der plattformspezifischen Modelle ist es, die Lücke, die zwischen dem plattformunabhängigen, abstrakten Anwendungsmodell und dem generierten Code besteht, zu minimieren. Die plattformspezifischen Modelle enthalten deshalb technische, implementierungsnahe Details einer Plattform, die zum Verständnis des Aufbaus des generierten Codes benötigt werden. Die Generierung der plattformspezifischen Modelle aus dem plattformunabhängigen Modell ist durch Modell-zu-Modell-Transformationen (M2M) mit QVTo[1] [71] implementiert. Die plattformspezifischen Modelle sind in Kapitel A erläutert.

Aus den plattformspezifischen Modellen wird der Quellcode für die Anwendung erzeugt. Aus dem plattformspezifischen Smart Card-Modell (Smart Card-PSM) wird der auf Chipkarten lauffähige Smart Card-Code erzeugt, aus dem Terminal-PSM wird der Code für die Terminals generiert. Für die Smart Cards wird Java Card-Code [90] generiert. Dies ist ein Java-Dialekt, der für ressourcenbeschränkte Geräte wie Smart Cards optimiert ist. Für die Terminals wird Java-Code erzeugt. Auch dieser Schritt erfolgt vollautomatisch, eine Ergänzung oder Änderung des Codes von Hand ist nicht nötig. Die Generierung des Quellcodes ist durch Modell-zu-Text-Transformationen (M2T) realisiert. Diese sind in XPand[2] implementiert. Die Generierung des Quellcodes ist in Kapitel 6 beschrieben.

Um auf einfache und schnelle Weise Fehler in der modellierten Anwendung zu finden, unterstützt der Ansatz außerdem das modellbasierte Testen der in der Entwicklung befindlichen Anwendung. Es werden sowohl funktionale Tests (zum Beispiel der Test, ob der Kontostand auf der Kopierkarte entsprechend des eingeworfenen Betrags erhöht wird) als auch Sicherheitstests (zum Beispiel der Test, ob der Angriff des Wiedereinspielens einer Nachricht mit einer Nonce zu einem späteren Zeitpunkt erfolgreich ist) unterstützt. Die Testfälle werden ebenfalls im plattformunabhängigen Modell auf abstrakter Ebene mit UML modelliert. Aus den model-

[1] Operational QVT Language for model to model transformations, `http://www.eclipse.org/m2m/`

[2] XPand: statically-typed template language for model to text transformations, `http://www.eclipse.org/modeling/m2t/?project=xpand`

lierten Testfällen wird automatisch Code generiert, der den zuvor generierten Quellcode der Anwendung testet und das Ergebnis des Testdurchlaufs graphisch darstellt. Die Generierung des Quellcodes für die Testfälle aus dem plattformunabhängigen Anwendungsmodell ist durch Modell-zu-Text-Transformationen mit XPand realisiert. Das modellbasierte Testen einer mit SecureMDD entwickelten Anwendung ist in Kapitel 7 beschrieben.

Das Testen der modellierten Anwendung bzw. des generierten Codes findet mit wenig Aufwand Protokollfehler und Sicherheitslücken. Allerdings ist es nicht möglich, die Abwesenheit von Fehlern und Sicherheitslücken sicherzustellen. Insbesondere Anwendungen, die auf kryptographischen Protokollen basieren, sind sehr anfällig für Fehler [5]. Um zu garantieren, dass eine Anwendung beim Entwurf definierte Sicherheitseigenschaften erfüllt, ist die Integration von formalen Methoden deshalb essentiell. Mit dem SecureMDD-Ansatz ist es möglich, aus dem plattformunabhängigen Modell der Anwendung automatisch eine formale Spezifikation zu generieren. Diese besteht aus algebraischen Spezifikationen und Abstract State Machines (ASM) [25, 77] und enthält das vollständige Anwendungsmodell inklusive der Protokolllogik. Die Spezifikation kann automatisch vom interaktiven Theorembeweiser KIV [9] eingelesen werden. Mithilfe des Tools werden die formulierten Sicherheitseigenschaften dann interaktiv verifiziert. Es können sowohl anwendungsunabhängige als auch anwendungsspezifische Eigenschaften verifiziert werden. Die Verifikation ist ein optionaler Teil des Ansatzes, d.h. es ist ebenfalls möglich, eine Anwendung mit UML zu modellieren und anschließend nur den Code zu generieren und diesen zu testen. Die Generierung der formalen Spezifikation sowie die Verifikation von Sicherheitseigenschaften für eine Anwendung sind in Kapitel 8 und 9 beschrieben.

Wird beim Testen oder der Verifikation einer Anwendung ein Fehler oder eine Sicherheitslücke entdeckt, muss das plattformunabhängige Anwendungsmodell entsprechend korrigiert werden. Da der gesamte Ansatz toolunterstützt ist, ist es mit wenigen Handgriffen möglich, den entsprechenden Anwendungscode (und falls nötig auch den Code für die Testfälle) neu zu generieren und die Testfälle erneut laufen zu lassen. Parallel dazu ist ein Neugenerieren der formalen Spezifikation und das Laden dieser in das Beweissystem ebenfalls ohne großen Aufwand möglich. Eventuell schon vorhandene Beweise bleiben weiterhin gültig, sofern sie nicht durch die im Modell getätigten Änderungen beeinflusst werden. Dies ist durch das Korrektheitsmanagement des Beweissystems sichergestellt.

Damit die auf dem formalen Modell verifizierten Sicherheitseigenschaften auch für den generierten Code gelten, muss der Code eine Verfeinerung der formalen Spezifikation sein. Bei der Erstellung der Modelltransformationen wurde deshalb darauf geachtet, dass der Code alle Anforderungen hierfür erfüllt (d.h. Code und formales Modell haben z.B. dieselbe Struktur und dasselbe Fehlerverhalten). In Kapitel 10 erfolgt eine nicht-formale Betrachtung der Eigenschaften des generierten Codes und der formalen Spezifikation, die für die Verfeinerungsbeziehung wichtig sind.

Das SecureMDD-Projekt

Die in dieser Dissertation vorgestellte Forschungsarbeit ist Teil des von der Deutschen Forschungsgemeinschaft (DFG)[3] geförderten Projekts SecureMDD. Neben dem zuvor vorgestellten Ansatz befasst sich das Projekt auch mit der Frage nach der formalen Korrektheit der

[3] www.dfg.de

erstellten Transformationen und erweitert den Ansatz um die Entwicklung von Serviceanwendungen. Diese beiden Punkte sind nicht Teil dieser Arbeit, werden aber der Vollständigkeit halber trotzdem im Folgenden kurz zusammengefasst.

Formaler Korrektheitsnachweis für die Transformationen Für eine konkrete Anwendung ist es möglich, die Verfeinerungsbeziehung zwischen dem Code der Anwendung und dem formalen Modell formal nachzuweisen. Dies wurde in dem Projekt *GoCard*, das dem SecureMDD-Projekt vorausging, gezeigt [65, 66, 167]. Die Verfeinerung muss dabei jedoch für jede Anwendung erneut und von Hand gezeigt werden. Da dabei der vollständige Code verifiziert werden muss, ist dies sehr aufwändig [66]. Ein Ziel des SecureMDD-Projekts ist es, formal nachzuweisen, dass die implementierten Modelltransformationen diese Verfeinerungsbeziehung immer, d.h. für alle mit dem Ansatz entwickelbaren Anwendungen, garantieren. Gelingt dies, ist sichergestellt, dass der generierte Code immer eine Verfeinerung des (ebenfalls generierten) formalen Modells ist und die auf dem Modell bewiesenen Sicherheitseigenschaften auch für den Code gelten. Es ist dann garantiert, dass der mit dem SecureMDD-Ansatz generierte Quellcode korrekt und sicher ist. Ein großer Vorteil dabei ist die erreichte Komplexitätsreduktion im Bereich der Verifikation. Beweist man die Sicherheitseigenschaften auf dem formalen Modell und gilt die Verfeinerungsbeziehung zwischen der formalen Spezifikation und dem Code, ist keine Quellcodeverifikation nötig.

Entwicklungen von Serviceanwendungen Das SecureMDD-Projekt befasst sich außerdem mit der Entwicklung von Anwendungen, die auf service-orientierten Architekturen (SOA) basieren. Im Fokus stehen dabei Business- sowie eGovernment-Anwendungen. Bei service-orientierten Architekturen wird die Funktionalität der Anwendung als austauschbare Services realisiert, die in verschiedenen Kontexten wiederverwendet sowie zu komplexen Systemen orchestriert werden können. Viele Business- und nahezu alle eGovernment-Anwendungen sind sicherheitskritisch. Für die meisten der entwickelten Anwendungen müssen deshalb ebenfalls, wie bei der Entwicklung von Smart Card-Anwendungen, anwendungsspezifische kryptographische Protokolle entwickelt werden. Beispiele für service-orientierte Anwendungen sind eVoting-Systeme (bei denen sichergestellt sein muss, dass ein Angreifer die abgegebenen Stimmen nicht manipulieren kann) oder Bankanwendungen, bei denen die Zahlungsabwicklung zwischen verschiedenen Banken durch Services realisiert ist (und z.B. sichergestellt sein muss, dass auch im Falle eines Angriffs der bei einer Transaktion abgebuchte Geldbetrag auf dem Zielkonto gutgeschrieben wird). Weitere Beispiele sind im Bereich eGovernment zu finden, z.B. die Ummeldung des Wohnsitzes beim Umzug in eine andere Stadt. Dabei muss sichergestellt sein, dass jede Person bei genau einer Meldebehörde registriert ist (in Deutschland hat jede Stadt eine eigene Meldebehörde, bei der die Bewohner dieser Stadt registriert sind). Natürlich spielt dabei in den meisten Anwendungen auch die Datensicherheit eine Rolle. Bei der Erweiterung des in dieser Arbeit vorgestellten Ansatzes um SOA-Anwendungen sind einige Punkte zu berücksichtigen. Speziell sind dies die Unterstützung von WS-Security [142] (ein OASIS-Standard[4], der kryptographische Operationen und Standardprotokolle für Serviceanwendungen zur Verfügung stellt), die Anbindung von externen Services sowie die Integration eines Sessionkonzepts, mit dem die komplexe Aufrufstruktur von Serviceanwendungen behandelt werden kann. Prinzipiell zeigt die mögliche Unterstützung für SOA-Anwendungen jedoch,

[4] https://www.oasis-open.org/

dass der bestehende Ansatz gut erweiterbar ist und in Zukunft auch andere Arten von Anwendungen, die auf kryptographischen Protokollen basieren, mit dem Ansatz entwickelt werden können. Hierfür ist eine Erweiterung des definierten UML-Profils, eventuell der domänenspezifischen Sprache sowie eine Anpassung der Modelltransformationen notwendig. Eventuell ist auch eine Erweiterung der formalen Spezifikation und der Beweistechnik notwendig.

3.3. Betrachtete Fallstudien

In diesem Abschnitt werden die im Rahmen der Arbeit durchgeführten Fallstudien vorgestellt. Abschnitt 3.3.1 erläutert die Kopierkartenanwendung. Da diese Fallstudie in den nachfolgenden Kapiteln zur Illustration verwendet wird, ist sie ausführlich beschrieben. Abschnitt 3.3.2 gibt einen Überblick über die Fallstudie „Elektronische Gesundheitskarte“. Diese ist im Detail in Abschnitt 11.2 erläutert. Abschließend werden in Abschnitt 3.3.3 alle weiteren Fallstudien kurz vorgestellt.

3.3.1. Fallstudie Kopierkartenanwendung

Zur Illustration wird in den folgenden Kapiteln eine Anwendung zum Anfertigen von Kopien, zum Beispiel innerhalb einer Universität, betrachtet. Die Nutzer des Kopierservices, beispielsweise Studenten, kaufen eine Smart Card. Diese Karte kann an einem Automaten durch den Einwurf von Geld aufgeladen werden. Hierfür wird die Karte in ein Lesegerät gesteckt, das sich in dem Automaten befindet. In der Universität gibt es außerdem Kopiergeräte, die ebenfalls mit einem Lesegerät ausgestattet sind. Möchte ein Benutzer Kopien anfertigen, steckt er seine Kopierkarte in das Lesegerät und gibt an, wie viele Kopien er machen möchte. Der entsprechende Betrag wird dann von der Karte abgebucht und die gewünschte Zahl an Kopien angefertigt. Außerdem ist es dem Benutzer möglich, am Ladeautomaten sowie am Kopiergerät den aktuellen Kontostand der Kopierkarte abzufragen. Die Kopierkartenanwendung ist ein einfaches, übersichtliches Beispiel, das in dieser Arbeit zur Erläuterung der Konzepte verwendet wird. Die Anwendung ist ähnlich zu der Geldkarte[5], allerdings nicht so sicher wie diese.

Jemand, der das Kopierkartensystem angreifen möchte, könnte versuchen, die Kommunikation zwischen den Komponenten des Systems zu beeinflussen. In dem Beispiel wird davon ausgegangen, dass die Kommunikation zwischen der Kopierkarte und dem Ladeautomaten sowie zwischen der Kopierkarte und dem Kopiergerät von einem Angreifer abgehört werden kann. Weiterhin kann ein Angreifer diese Kommunikation manipulieren, d.h. Nachrichten verändern und austauschen sowie Nachrichten unterdrücken. Die Kommunikation zwischen dem Kartenbesitzer und dem Automat zum Aufladen bzw. dem Kopiergerät kann ein Angreifer nicht beeinflussen. Unter diese Kommunikation fallen die Eingaben, die der Kartenbesitzer (über eine graphische Benutzeroberfläche) am Automaten bzw. Kopiergerät macht. Zum Beispiel teilt der Kartenbesitzer dem Ladeautomaten mit, wie viele Cent er gerne auf die Karte laden möchte. Außerdem gibt der Ladeautomat dem Kartenbesitzer am Ende des Ladevorgangs eine Statusmeldung, ob der Vorgang erfolgreich war oder nicht. Der Angreifer ist nicht zwangsläufig eine dritte, außenstehende Person, sondern möglicherweise ein Kartenbesitzer,

[5] www.geldkarte.de

der versucht sich einen (wirtschaftlichen) Vorteil zu verschaffen. Die Kommunikationskanäle sowie die Möglichkeiten des Angreifers sind in Abbildung 3.3 dargestellt.

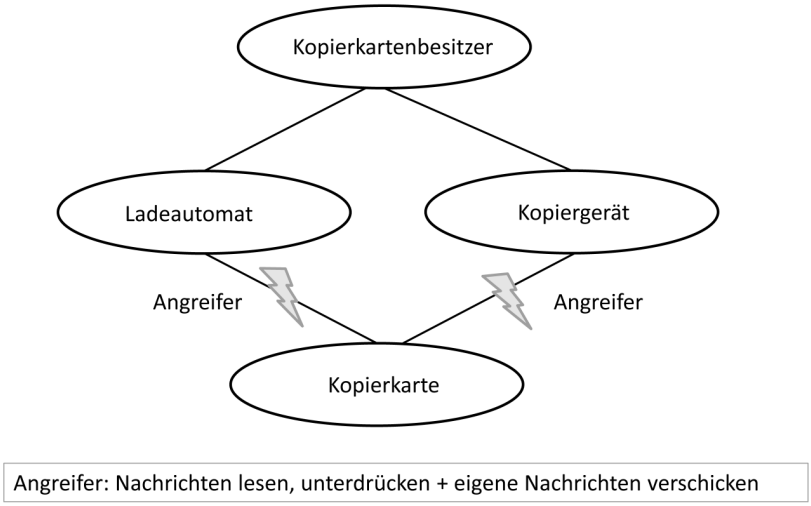

Abbildung 3.3.: Komponenten der Kopierkarte mit Angreifer

Der Betreiber des Kopierservices möchte, dass die Anwendung sicher ist gegenüber einem möglichen Angreifer. Dieser könnte zum Beispiel versuchen, Kopien anzufertigen, ohne dass dafür Geld von seiner Karte abgebucht wird. Oder er könnte versuchen, Geld auf eine (echte) Kopierkarte zu laden, ohne dafür Geld in den Automaten einzuwerfen. Dies könnte er erreichen, indem er die Karte an seinem PC zu Hause auflädt. Eine weitere Angriffsmöglichkeit wäre, dass der Angreifer sich eine eigene Kopierkarte programmiert und mit dieser versucht, am Kopiergerät Kopien anzufertigen. Es liegt im Interesse des Kopierkartenbetreibers, dass solche Angriffe nicht möglich sind und der Betreiber somit kein Geld verliert.

Das Ziel ist also, eine Anwendung zu entwickeln, die diese Sicherheitsanforderungen erfüllt. Um dies zu erreichen, müssen (kryptographische) Protokolle entworfen werden, die die Kommunikation zwischen allen an dem System Beteiligten festlegen.

Beim Design der Protokolle für die Kopierkartenwendung müssen folgende Aspekte berücksichtigt werden:

- Beim Aufladen der Karte muss sich das Ladeterminal gegenüber der Kopierkarte als echt beweisen. Dies stellt sicher, dass nur Geld auf eine Kopierkarte geladen wird, wenn zuvor der entsprechende Geldbetrag in ein echtes, zum Betreiber gehörendes, Ladeterminal eingeworfen wurde. Beim Bezahlen muss sich die Kopierkarte gegenüber dem Kopiergerät als echt beweisen. Auf diese Weise ist sichergestellt, dass Kopien nur herausgegeben werden, wenn der entsprechende Geldbetrag zuvor von einer echten Kopierkarte abgebucht wurde.
- Es muss verhindert werden, dass Nachrichten vom Angreifer gelesen, gespeichert und zu einem späteren Zeitpunkt erfolgreich wiedereingespielt werden können (Replay Angriff). Dies muss auch protokollübergreifend, also für Nachrichten bzw. Teile von Nachrichten zwischen verschiedenen Protokollen, ausgeschlossen werden.

- Der Betrag, der auf die Karte geladen wird, muss mit dem Geldbetrag übereinstimmen, den der Kartenbesitzer in Form von Bargeld in das Ladeterminal wirft. Werden Kopien angefertigt, muss der Wert der Kopien genau dem Betrag, der von der Kopierkarte abgebucht wird, entsprechen.
- Die Kopierkarte soll ein Ersatz für Bargeld sein. Verliert man die Karte, ist somit auch der noch auf der Karte gespeicherte Wert für Kopien verloren. Der Finder der Karte kann diese benutzen und Kopien anfertigen. Es gibt keinen Schutz, der dies verhindert. Die Alternative wäre eine personalisierte Karte, die durch eine PIN geschützt ist. Damit ist es nur demjenigen, der die PIN kennt, möglich, Kopien anzufertigen.
- Die Protokolle sollen sicherstellen, dass der Betreiber der Kopierkartenanwendung nicht von einem seiner Nutzer betrogen werden kann. Um die Protokolle übersichtlich zu halten, wird die Sicherheit für den Benutzer jedoch vernachlässigt. So ist es beispielsweise möglich, dass beim Zahlen von Kopien der entsprechende Betrag schon von der Kopierkarte abgebucht wird, die Rückantwort der Karte jedoch von einem Angreifer abgefangen wird. In diesem Fall verliert der Kartenbesitzer Geld, da das Kopiergerät die Kopien nicht freigibt.

Um die Protokolle zu verstehen, ist es notwendig, zuvor einige Begriffe zu erläutern.

Eine *Nonce* ist eine Zufallszahl, die nur einmal in einem Kontext verwendet wird. Eine zuvor noch nicht verwendete Nonce wird auch als *fresh* bezeichnet. Eine Nonce wird beispielsweise zum Verhindern von Replay-Angriffen, d.h. dem späteren Wiedereinspielen einer Nachricht, verwendet. Ein weiteres Beispiel für die Verwendung von Nonces ist die Authentisierung einer Komponente, die in Kryptoprotokollen oft durch sogenannte *Challenge-Response-Verfahren* implementiert ist. Bei diesen weist die Komponente, die sich authentisieren muss, nach, dass sie einen geheimen Wert oder einen geheimen Schlüssel kennt und deshalb diejenige ist, die sie zu sein vorgibt. In den Protokollen der Kopierkarte wird eine Komponente als authentisch angesehen, wenn sie ein bestimmtes Geheimnis kennt. Dieses Geheimnis ist allen echten Kopierkarten sowie allen Terminals bekannt. Der Nachteil dieses Vorgehens ist, dass das gesamte System (d.h. alle ausgegebenen Kopierkarten sowie alle Terminals) betroffen ist, sollte ein Angreifer dieses Geheimnis erfahren.

Eine *Hashfunktion* ist eine Funktion, die eine Eingabe beliebiger Länge in eine Ausgabe fester (im allgemeinen kleinerer) Länge konvertiert. Ein *Hashwert* wird durch Anwendung einer Hashfunktion erzeugt. Gute Hashfunktionen sind kollisionsfrei. Das bedeutet, dass zwei Daten (auch wenn sie nur sehr geringe Unterschiede aufweisen) mit sehr hoher Wahrscheinlichkeit unterschiedliche Hashwerte besitzen. Außerdem ist es sehr schwierig, zwei Daten zu finden, die denselben Hashwert haben. Man sagt deshalb, der Hashwert sei der Fingerabdruck eines Datums. Hashfunktionen sind Einweg-Funktionen, d.h. es ist einfach, den Hashwert eines Datums zu berechnen. Zu einem gegebenen Hashwert das zugehörige Datum zu finden ist jedoch praktisch unmöglich. Eine detaillierte Erläuterung weiterer kryptographischer Primitive sowie ihre Verwendung wird z.B. von Schneier in [171] gegeben.

Die als Fallstudie dienende Kopierkartenanwendung verwendet Hashwerte und kommt ohne Verschlüsselung und digitale Signaturen aus. Alle echten, also vom Serviceanbieter ausgegebenen Karten sowie alle Automaten zum Aufladen und Bezahlen, kennen ein gemeinsames Geheimnis. Dieses ist für alle gleich und für die Sicherheit der Anwendung ist es wichtig, dass ein Angreifer dieses Geheimnis nicht erfährt.

Im Folgenden sind die Protokolle der Kopierkartenanwendung beschrieben, die die o.g. Sicherheitsanforderungen erfüllen. Die Protokolle sind mit Sequenzdiagrammen modelliert. Diese enthalten nur die Nachrichten, die zwischen den Beteiligten ausgetauscht werden. Das interne Verhalten, wie zum Beispiel das Ändern des Kontostandes auf der Karte, ist nicht modelliert. Dies entspricht der in der Literatur klassischerweise verwendeten Darstellungsweise von kryptographischen Protokollen [4, 5, 28]. Die vollständigen Protokolle, einschließlich des internen Verhaltens, werden in Abschnitt 4.5.2 vorgestellt.

Abbildung 3.4 zeigt das Protokoll zum Laden eines Geldbetrags auf eine Kopierkarte. Die an diesem Protokoll beteiligten Komponenten sind die Studenten, die eine Kopierkarte besitzen (`CardOwner`), die Automaten zum Aufladen der Karten (`DepositMachine`), die Kopiergeräte (`CopyingMachine`) sowie die Kopierkarten (`Copycard`).

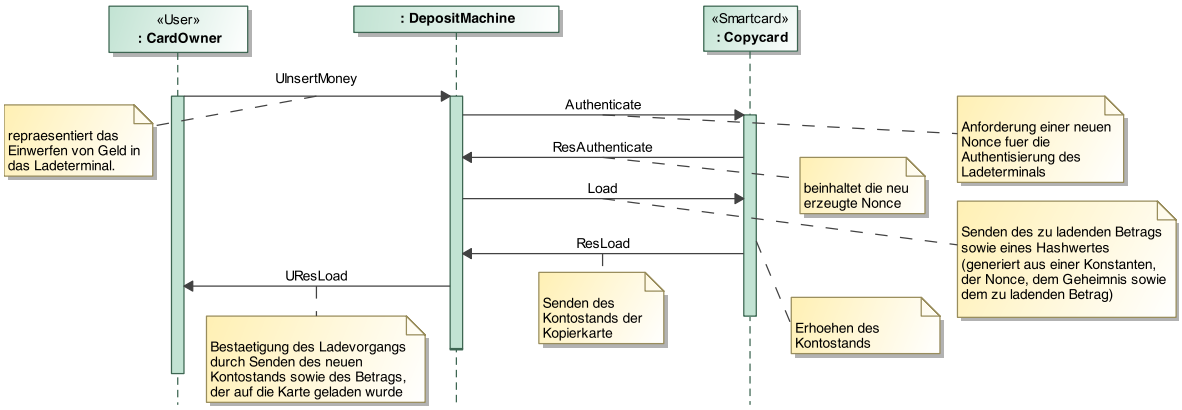

Abbildung 3.4.: Protokoll zum Laden eines Geldbetrags auf die Kopierkarte

Das Protokoll beginnt damit, dass der Kartenbesitzer einen Geldbetrag (mit Wert `amountToLoad`) in das Ladeterminal einwirft und dem Ladeterminal (über eine Benutzeroberfläche) mitteilt, dass er das Geld auf die Karte laden möchte. Dies ist modelliert durch das Senden der Nachricht `UInsertMoney(amountToLoad)` an das Ladeterminal `DepositMachine`. Das Laden von Geld auf die Kopierkarte soll nur mit einem echten Ladeterminal, also einem, das das gemeinsame Geheimnis kennt, möglich sein. Aus diesem Grund ist es notwendig, dass sich das Ladeterminal bei der Kopierkarte authentisiert, bevor der Betrag auf der Karte um den zu ladenden Betrag inkrementiert wird. Hierfür sendet das Ladeterminal eine Nachricht `Authenticate()` an die Kopierkarte, mit der es den Authentisierungsvorgang initiiert. Die Authentisierung geschieht mittels eines *Challenge-Response-Verfahrens*, d.h. die Kopierkarte erzeugt eine Nonce (mit dem Namen `challenge`) und sendet diese (in der Nachricht `ResAuthenticate(challenge)`) an das Ladeterminal. Das Ladeterminal berechnet den Hashwert aus der Konstanten `LOAD`, der empfangenen Nonce, dem gemeinsamen Geheimnis sowie dem Betrag, der auf die Karte geladen werden soll. Dieser Hashwert (mit dem Namen `authterminal`) wird dann zusammen mit dem zu ladenden Betrag `amountToLoad` in der Nachricht `Load(amountToLoad, authterminal)` an die Kopierkarte geschickt. Hashwerte haben die Eigenschaft, dass es nicht möglich ist, aus dem Wert wieder die ursprünglichen Daten zu errechnen. Einem Angreifer ist es also nicht möglich, beim Lesen der Nachricht das Geheimnis zu bestimmen. Die Karte empfängt die

Nachricht und bildet ebenfalls einen Hashwert aus der Konstanten, der im vorherigen Schritt generierten Nonce, dem Geheimnis und dem zu ladenden Betrag `amountToLoad`. Stimmen die Hashwerte überein, weiß die Kopierkarte, dass das Ladeterminal echt ist (da es das Geheimnis kennt), die Nachricht nicht aus einem vorherigen Protokolllauf stammt (da die im Hashwert enthaltene Nonce neu erzeugt wurde und nur für den aktuellen Ladeschritt verwendet wird) und der übertragene Betrag korrekt ist (weil er ebenfalls im Hashwert enthalten ist und somit nicht verändert wurde). Die Sicherheit des Protokolls liegt somit im Wesentlichen in der Verwendung des Hashwertes. Die Karte akzeptiert daraufhin den Ladevorgang und inkrementiert den Kontostand entsprechend des Betrags. Als Antwort sendet sie eine Nachricht `ResLoad` zurück an das Ladeterminal, die den neuen Kontostand der Karte beinhaltet (`ResLoad(balance)`). Das Ladeterminal bestätigt dem Kartenbesitzer den Ladevorgang in einer `UResLoad`-Nachricht, in der der Benutzer zusätzlich den neuen Kontostand `balance` sowie Betrag `amountToLoad` erfährt.

Nun wird noch der Vorgang des Anfertigens von Kopien betrachtet, der in Abbildung 3.5 dargestellt ist.

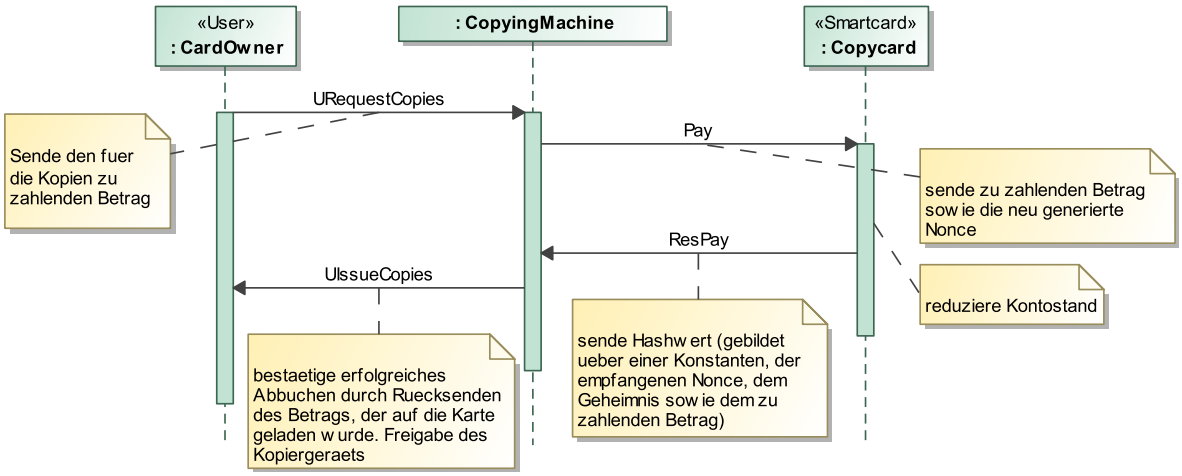

Abbildung 3.5.: Protokoll zum Abbuchen eines Geldbetrags von der Kopierkarte

Das Protokoll beginnt damit, dass der Kartenbesitzer angibt, wie viele Kopien bzw. für welchen Betrag er Kopien anfertigen möchte (Nachricht `URequestCopies(amountToPay)` an das Kopiergerät `CopyingMachine`). Beim Anfertigen von Kopien ist es wichtig, dass Kopien nur mit einer echten Kopierkarte bezahlt werden können. Ansonsten wäre es möglich, dass ein Angreifer eine selbstprogrammierte Karte verwendet und so Kopien machen kann, ohne für diese zu zahlen. Die Authentifizierung der Karte funktioniert nach dem gleichen Prinzip wie die Authentisierung des Ladeterminals beim Aufladen der Karte. Das Kopiergerät generiert eine neue Nonce `challenge` und sendet diese zusammen mit dem abzubuchenden Betrag in der Nachricht `Pay(amountToLoad, challenge)` an die Kopierkarte. Diese bildet den Hashwert über der Konstanten `PAY`, der empfangenen Nonce `challenge`, dem gemeinsamen Geheimnis sowie dem zu ladenden Betrag `amountToPay`. Anschließend reduziert die Karte ihren Kontostand um den Wert `amountToPay` und sendet den Hashwert zurück an das Kopiergerät (Nachricht `ResPay(authcard)`). Das Kopiergerät überprüft den empfangenen Hashwert und fertigt die Kopien an, wenn der Hashwert in Ordnung ist, d.h. wenn der Hash-

wert über der Konstanten PAY, der zuvor generierten Nonce, dem gemeinsamen Geheimnis und dem zu zahlenden Betrag gebildet wurde. Als Antwort an den Kartenbesitzer sendet das Kopiergerät eine `UIssueCopies` Nachricht an diesen zurück, die zur Bestätigung den bezahlten Betrag `amountToPay` enthält.

Ein weiteres Protokoll der Kopierkartenanwendung ermöglicht das Abfragen des Kontostands der Karte und ist in Abbildung 3.6 dargestellt.

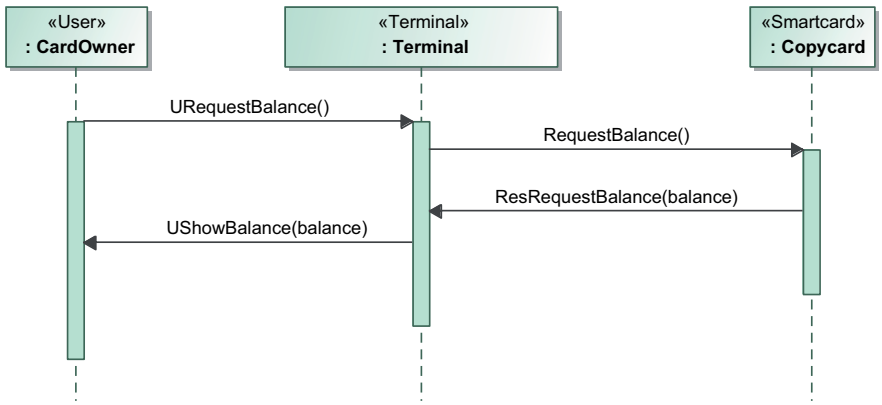

Abbildung 3.6.: Protokoll zum Abfragen des Kontostands der Kopierkarte

Die Abfrage des Kontostands ist sowohl an einem Ladeterminal als auch an einem Kopiergerät möglich. Im Sequenzdiagramm wird deshalb als Kommunikationspartner eine abstrakte Komponente `Terminal` verwendet, die sowohl ein Ladeterminal als auch ein Kopiergerät repräsentiert. Dies soll verdeutlichen, dass die Funktion „Abfragen des Kontostands“ an beiden Gerätetypen zur Verfügung steht.

Der Kartenbesitzer möchte wissen, welcher Geldbetrag noch auf seiner Karte gespeichert ist. Er schickt deshalb (über eine Benutzeroberfläche) eine `URequestBalance` Nachricht an das Terminal. Dieses leitet die Anfrage in einer `RequestBalance` Nachricht an die Kopierkarte weiter. Die Kopierkarte antwortet dem Terminal mit einer `ResRequestBalance` Nachricht, die den aktuellen Kontostand `balance` enthält. Diesen leitet das Terminal in einer `UResRequestBalance` Nachricht weiter an den Kartenbesitzer.

An dieser Stelle wird wieder ersichtlich, dass die vorgestellten Protokolle lediglich die Sicherheitswünsche des Kopierservicebetreibers erfüllen und nicht die des Kunden. Einem Angreifer ist es möglich die Verbindung zwischen Terminal und Kopierkarte abzuhören und so den Kontostand der Kopierkarte zu erfahren. Dies widerspricht dem Schutz von vertraulichen Daten. Außerdem kann ein Angreifer die `URequestBalance` Nachricht manipulieren und den übertragenen Kontostand `balance` ändern, so dass dem Kartenbesitzer ein falscher Wert mitgeteilt wird. Diese Probleme lassen sich lösen, indem die `ResRequestBalance` Nachricht verschlüsselt und gegen Manipulation durch Hashwerte abgesichert wird. Um die Fallstudie übersichtlich zu halten, wurde hierauf jedoch verzichtet.

Die vorgestellten Protokolle sollen verdeutlichen, dass das Design von kryptographischen Protokollen und somit der Entwurf von sicherheitskritischen Anwendungen schwierig, die Angriffsmöglichkeiten auf die Protokolle nicht immer gleich ersichtlich und die Protokolle deshalb sehr

fehleranfällig sind. Auch für kleinere Anwendungen wie das Kopierkartenbeispiel ist es keinesfalls trivial, korrekte und die Sicherheitseigenschaften erfüllende Protokolle zu entwerfen.

3.3.2. Fallstudie „Elektronische Gesundheitskarte“

Die deutsche elektronische Gesundheitskarte (EGK) ist ein Beispiel für eine sehr große und facettenreiche Fallstudie. Verantwortlich für die Entwicklung dieser Anwendung ist die gematik[6], die die EGK ursprünglich schon im Jahr 2006 in Deutschland einführen wollte. Aufgrund von offenen Fragen, insbesondere bezüglich der Datensicherheit sowie der Übernahme der Kosten, wurde die Einführung jedoch immer wieder verzögert. Zurzeit wird die elektronische Gesundheitskarte mit eingeschränkter Funktionalität in Deutschland ausgegeben. Bis Ende des Jahres 2012 ist vorgesehen, dass 50% der gesetzlich Versicherten die neue elektronische Gesundheitskarte besitzen. Ihre Funktionalität beschränkt sich jedoch auf die Speicherung der Daten, die auch auf der bisherigen Karte vorhanden sind sowie den Aufdruck eines Fotos zur Identifikation des Kartenbesitzers. Weitere Funktionalität ist als Erweiterung für die Zukunft geplant.

Im Rahmen dieser Arbeit wurde die elektronische Gesundheitskarte als Fallstudie untersucht. Da die meisten von der gematik entworfenen Protokolle und Anwendungen zurzeit obsolet sind, wurden dafür von uns eigene Protokolle entworfen und umgesetzt. Die unterstützte Funktionalität umfasst dabei:

- das Erstellen eines elektronischen Rezepts, das von einem Heilberufsausweis eines Arztes signiert und anschließend auf der elektronischen Gesundheitskarte gespeichert wird
- das Einlösen eines elektronischen Rezepts
- das Erstellen und Ändern von Notfalldaten (wie zum Beispiel Unverträglichkeiten oder Diabetes) eines Patienten, die auf seiner elektronischen Gesundheitskarte gespeichert werden
- das Einsehen der auf der elektronischen Gesundheitskarte gespeicherten Daten durch den Patienten an einem Kiosk-System, das in Apotheken aufgestellt ist
- das Löschen der Notfalldaten auf einer EGK durch den Patienten am Kiosk-System. Das Speichern der Notfalldaten ist optional

Die an der Anwendung beteiligten Personengruppen sind Ärzte, Apotheker und Patienten. Alle drei Gruppen besitzen eine Smart Card. Die Ärzte und Apotheker authentisieren sich über den Besitz ihrer Heilberufsausweise sowie die Eingabe einer PIN-Nummer. Die Patienten besitzen eine Gesundheitskarte, für die sie ebenfalls eine PIN-Nummer besitzen und auf der medizinische Daten gespeichert werden. Weiterhin gibt es drei verschiedene Arten von Terminals: Die Arzt-PCs, die in den Arztpraxen stehen, die Apotheken-PCs sowie Kiosksysteme, die in den Apotheken stehen. Mithilfe der Kiosksysteme kann ein Patient Einsicht in seine auf der Gesundheitskarte gespeicherten Daten erhalten.

Die Fallstudie ist wesentlich komplexer (in Bezug auf die beteiligten Komponenten und ihr Zusammenspiel sowie auch die Anzahl und Komplexität einzelner Protokolle) als die anderen betrachteten Fallstudien. Sie umfasst 15 Protokolle mit insgesamt 105 Protokollschritten. Aus dem plattformunabhängigen UML-Modell wurden ca. 83.500 Zeilen (teilweise redundanter)

[6] Gesellschaft für Telematikanwendungen der Gesundheitskarte, `http://www.gematik.de`

Code erstellt. Anhand des Beispiels konnte gezeigt werden, dass der modellgetriebene Ansatz auch für die Entwicklung großer und komplexer Anwendungen gut geeignet ist. Detailliert wird auf diese Fallstudie in Kapitel 11.2 eingegangen.

3.3.3. Weitere Fallstudien

Der SecureMDD-Ansatz wurde für die Kopierkartenanwendung sowie die elektronische Gesundheitskarte vollständig durchlaufen, einschließlich der Codegenerierung sowie Verifikation. Außerdem wurde der Ansatz anhand von vier weiteren Fallstudien evaluiert und konsolidiert. Dabei wurden sowohl eigene kryptographische Protokolle entworfen als auch bereits bestehende Protokolle verwendet.

Die vier Fallstudien sind im Folgenden kurz erläutert, die erstellten Modelle sowie alle weiteren Artefakte sind auf der Webseite des SecureMDD-Projekts[7] dargestellt.

- **Elektronischer Reisepass**: In dieser Fallstudie wurde die in der Praxis existierende Anwendung des elektronischen Reisepasses [58] mit dem SecureMDD-Ansatz nachmodelliert und lauffähiger Code generiert. Betrachtet wurde das *Basic Access Control*-Verfahren, das die Authentisierung zwischen einem Reisepass und einem entsprechenden Inspektionsgerät sowie den Austausch eines symmetrischen Sitzungsschlüssels für die Verschlüsselung des Datenaustauschs beinhaltet. Außerdem modelliert wurden die Protokolle zum Auslesen der auf dem Pass gespeicherten Daten. Diese verwenden symmetrische Verschlüsselung und Message Authentication Codes (MAC-Werte), um die Integrität der Daten sicherzustellen.
- **Geldkarte**: Es wurde eine Version der deutschen Geldkarte[8] betrachtet, die auf asymmetrischer Verschlüsselung und digitalen Signaturen basiert. Die Geldkarte wird als Ersatz für Bargeld verwendet und kann an einem Automaten aufgeladen werden.
- **Altersverifikation**: Die Altersverifikation ist eine Anwendung u.a. des deutschen Personalausweises und ermöglicht die Verifikation des Alters einer Person. Zum Beispiel ist der Zigarettenkauf an einem Automaten nur dann möglich, wenn der Käufer älter als 16 Jahre ist. Eine vereinfachte Version dieser Anwendung, basierend auf Zertifikaten, wurde mit dem vorgestellten Ansatz modelliert sowie lauffähiger Code generiert.
- **Mondex**: Mondex ist eine elektronische Geldbörse, die auf der Zahlung mit Chipkarten basiert. Beim Einkauf wird Geld von der Mondexkarte des Käufers auf die Mondexkarte des Verkäufers übertragen. Die Sicherheitseigenschaft, die durch das Mondex-Protokoll erfüllt ist, besagt, dass „niemals Geld verloren geht oder erzeugt werden kann“. Dies gilt auch dann, wenn ein Angreifer die Kommunikation beeinflusst und z.B. Nachrichten wiedereinspielt, selbst generierte Nachrichten sendet oder Nachrichten unterdrückt. Die Protokolle basieren auf symmetrischer Verschlüsselung. Die Mondexanwendung wurde mit dem SecureMDD-Ansatz modelliert, lauffähiger Quellcode erzeugt und dieser (mit modellbasierten Tests) getestet. Mondex war im Jahr 2006 als Grand Challenge für Theorembeweiser ausgeschrieben [80, 93, 169].

[7] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/`
[8] `https://www.geldkarte.de`

3.4. Verwandte Arbeiten

Es gibt viele verschiedene Forschungsarbeiten, die sich mit modellgetriebenen Ansätzen für die Entwicklung sicherer Anwendungen beschäftigen. [92] und [104] (sowie der etwas ältere Artikel von Devanbu [49]) geben einen guten Überblick über (einige) dieser Arbeiten.

Im Folgenden werden die für diese Dissertation wichtigen Arbeiten zum Thema „Model-Driven Security“ vorgestellt und mit dem SecureMDD-Ansatz verglichen.

UMLSec:
Jan Jürjens stellt mit UMLSec [20, 89, 99, 100] eine Methodik und ein Werkzeug zur Verfügung, um sicherheitskritische Anwendungen zu entwerfen und formal zu verifizieren. UMLSec unterstützt auch den Entwurf kryptographischer Protokolle und integriert formale Methoden, um Sicherheitseigenschaften für die entworfenen Anwendungen zu garantieren.

UMLSec erweitert die Unified Modeling Language um ein UML-Profil. Dieses definiert sowohl Stereotypen und Tags für die Modellierung der Anwendung selbst als auch für die Definition von Standardsicherheitseigenschaften (wie z.B. Geheimhaltung, Integrität und Authentizität von Daten, Informationsflusseigenschaften wie `no down-flow` und `no up-flow` und rollenbasierte Zugriffskontrolle (`rbac`)). Zum Beispiel gibt es den Stereotyp ≪critical≫, der eine Komponente annotiert, die für die Sicherheit einer Anwendung kritisch ist. Über verschiedene Tags kann dann genauer spezifiziert werden, welche Attribute und Assoziationen dieser Komponente in welcher Weise sicherheitskritisch sind. `{secrecy = s}` bedeutet zum Beispiel, dass der Wert des Attributs `s` geheim bleiben muss. Wie auch SecureMDD verwendet UMLSec verschiedene UML-Diagrammtypen für die Modellierung. Die Verwendung der Deploymentdiagramme ist ähnlich zu SecureMDD. Ein Unterschied ist jedoch, dass SecureMDD die Modellierung der kompletten Protokolle durch Aktivitätsdiagramme und die DSL MEL unterstützt. Dies umfasst auch die Modellierung der einzelnen Protokollschritte, d.h. zum Beispiel dem Abbuchen eines Geldbetrags von einer Kopierkarte. UMLSec verwendet für die dynamische Modellierung UML-Aktivitäts- und Sequenzdiagramme sowie Zustandsmaschinen. Ähnlich zu MEL definiert auch Jürjens eine Sprache, um kryptographische Protokolle modellieren zu können (in [89], Kapitel 3). Diese ist doch nicht in die UML-Diagramme integriert, sondern Ausdrücke der Sprache werden neben die UML-Diagramme geschrieben. Ziel ist es nicht, die Protokollschritte zu modellieren, sondern nur die Kommunikation, d.h. die ausgetauschten Nachrichten. Das heißt, es ist z.B. möglich, Teile einer empfangenen Nachricht zu entschlüsseln und anschließend zu versenden. Fallunterscheidungen, die das Verhalten der Anwendung bei Auftreten eines Fehlers oder Angriffs beschreiben oder arithmetische Operationen können nicht modelliert werden. Abbildung 3.7 zeigt das Sequenzdiagramm für das Aufladen der Smart Card in der von Jürjens untersuchten Fallstudie Common Electronic Purse (aus [89], Seite 105).

UMLSec ist mit einer formalen Semantik unterlegt, die in [89] beschrieben ist. UMLSec hat als Semantik sogenannte UML Machines und UML Machine Systems, die operational die durch die UML-Diagramme möglichen Abläufe einer Anwendung beschreiben.

Der Ansatz ist toolunterstützt. Das UMLSec-Tool verbindet das für die Modellierung verwendete Tool mit einer „Security Tool Suite“, in der automatische Theorembeweiser (z.B. e-SETHEO [139], SPASS [188] und Waldmeister [85]), der Model Checker Spin [86] und ein auf Prolog basierender automatischer Angriffsgenerator integriert sind. UMLSec wurde bereits

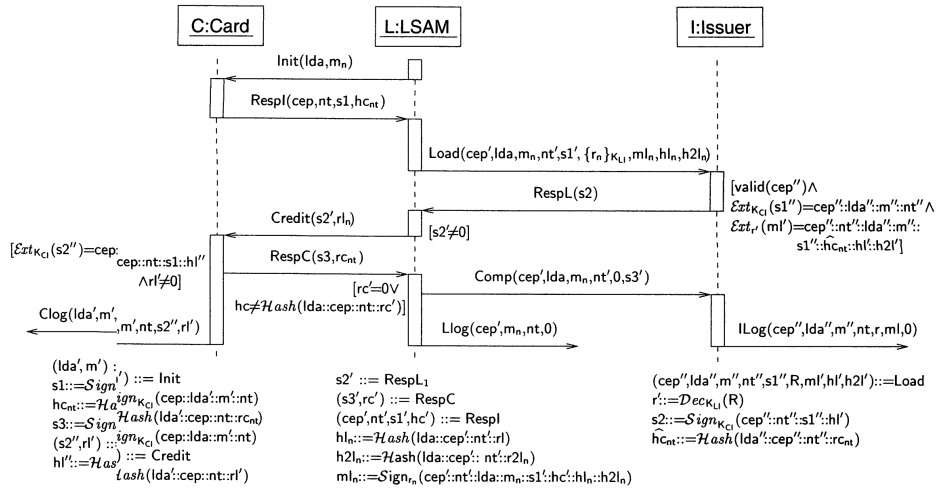

Abbildung 3.7.: Ausschnitt aus dem UMLSec-Modell für CEPS aus [89], Seite 105

für viele Fallstudien eingesetzt, u.a. im Bereich mobiler Anwendungen [99], im Automobilbereich, für Smart Card-Anwendungen oder für Internetprotokolle (alle in [89]).

Neben den genannten Gemeinsamkeiten gibt es auch einige Unterschiede zwischen UMLSec und SecureMDD. UMLSec ist beschränkt auf die Betrachtung von Standardsicherheitseigenschaften (wie z.B. Geheimhaltung und Integrität), während der SecureMDD-Ansatz sowohl die Verifikation dieser Eigenschaften als auch anwendungsspezifischer Eigenschaften unterstützt (siehe Kapitel 9). Anwendungsspezifische Eigenschaften sind abhängig von der konkreten Anwendung, z.B. dass der Betreiber der Kopierkartenanwendung kein Geld verliert oder ein Rezept, das auf einer Gesundheitskarte gespeichert ist, nicht zweimal in der Apotheke eingelöst werden kann. Meiner Meinung nach geben anwendungsspezifische Sicherheitseigenschaften für die in dieser Arbeit (und auch von Jürjens mit UMLSec) betrachteten Smart Card-Anwendungen bessere Garantien für die Sicherheit als Standardeigenschaften wie die Geheimhaltung einzelner Daten. Die Verifikation von Standardeigenschaften ist in der Regel eine Voraussetzung für die Verifikation von anwendungsspezifischen Eigenschaften und wird von SecureMDD ebenfalls unterstützt. Die formale Semantik vom UMLSec bleibt außerdem ohne Verifikationsunterstützung. Die verwendeten automatischen Beweistools enthalten nur einen Teil der durch die UML Machine Systems beschriebenen Abläufe. Bei SecureMDD dagegen ist die gesamte Anwendung im formalen Modell abgebildet und es werden alle möglichen Abläufe betrachtet.

UMLSec unterstützt die Modellierung einer sicherheitskritischen Anwendung mit UML sowie die Verifikation von Standardeigenschaften. Die Generierung lauffähigen Quellcodes aus dem UML-Modell ist mit UMLSec nicht möglich. Der SecureMDD-Ansatz dagegen ermöglicht die Generierung einer lauffähigen Implementierung der modellierten Anwendung, für die die auf formaler Ebene bewiesenen Sicherheitseigenschaften ebenfalls gelten. Dies ist ein großer Vorteil im Vergleich zum UMLSec-Ansatz. Dass es für die Sicherheit einer Anwendung essentiell ist, auch die Codeebene zu betrachten, wurde von Jürjens ebenfalls erkannt. UMLSec integriert deshalb einen Code-Checker, der handgeschriebenen Java-Code auf bestimmte Ei-

genschaften prüfen kann. Dies geschieht jedoch unabhängig von der Modellierung mit UML. Außerdem gibt es für die rollenbasierte Zugriffskontrolle die Möglichkeit, Code zu generieren, der die auf UML-Modell angegebenen RBAC-Eigenschaften sicherstellt [137].

Autofocus:
Autofocus [88] ist ein Werkzeug für die Entwicklung und Verifikation verteilter Systeme. Ähnlich zu der UML erlaubt Autofocus die Modellierung unterschiedlicher Sichten eines Systems. Dafür werden Datentypdefinitionen, Systemstrukturdiagramme, sowie State Transition-Diagramme verwendet. Außerdem wird die Modellierung einzelner Systemabläufe durch Extended Event Traces unterstützt. Autofocus hat einen Anschluss an den Model Checker SMV für die Verifikation von Eigenschaften für die modellierten Anwendungen. In [102] wird am Beispiel der „Common Electronic Purse“ (eine Spezifikation für eine währungsüberschreitende Elektronische Geldbörse) gezeigt, dass die Methodik auch für Smart Card-Anwendungen geeignet ist.

Gegenüber dem SecureMDD-Ansatz und der dort verwendeten interaktiven Verifikation hat Autofocus (wie auch alle anderen Model Checking-gestützten Ansätze zur Verifikation kryptographischer Protokolle) den Nachteil, dass die Anzahl der Protokollläufe und -teilnehmer fixiert werden muss, da der Zustandsraum sonst im Allgemeinen unendlich wird. Bei der interaktiven Verifikation ist diese Einschränkung nicht nötig. Ein Vorteil gegenüber SecureMDD ist jedoch, dass bei Nichtgelten einer Eigenschaft ein Message Sequence Chart generiert werden kann, das den gefundenen Angriff visualisiert. Autofocus ist gut für die Entwicklung eingebetteter Systeme geeignet [178] und unterstützt auch die Generierung von Quellcode (C,Java und Ada) für diese Systeme. Die Generierung von Code für Smart Card-Anwendungen bzw. kryptographische Protokolle ist jedoch nicht möglich.

Weitere Ansätze für die Entwicklung von Smart Card-Anwendungen:
Nikseresht et al. stellen in [145] einen modellgetriebenen Ansatz für die Entwicklung von Smart Card-Anwendungen vor. Die plattformunabhängige Modellierung erfolgt wie in dieser Arbeit in UML. Aus dem PIM wird zunächst ein plattformspezifisches Modell und aus diesem Smart Card-Code generiert. Der Ansatz ist an einer Krankenhausanwendung, bei der die Patientendaten auf einer Smart Card gespeichert sind, illustriert. Der Ansatz unterstützt nur die Modellierung der statischen Sicht einer Anwendung mit UML. Die kryptographischen Protokolle werden nicht betrachtet. Außerdem ist die Betrachtung der Sicherheit der entwickelten Anwendungen nicht in den Ansatz integriert. Es können somit keine Garantien bezüglich der Sicherheit einer Anwendung gegeben werden. Dies ist ein Unterschied zu dem in dieser Arbeit vorgestellten Ansatz.

Mostwoski stellt in [140] einen Ansatz für die systematische Entwicklung von Java Card-Applets vor. Dort wird ein Java Card-Applet mit UML-Zustandmaschinen modelliert. In dem UML-Modell sind nur die möglichen Zustände des Applets sowie die durch Senden einer Nachricht ausgelösten Zustandsübergänge modelliert. Die eigentlichen Protokollschritte, die das Applet ausführen kann, sind nicht Teil der Modellierung, sondern müssen von Hand im Code ergänzt werden. Außerdem kann das UML-Modell um OCL-Constraints ergänzt werden. Diese Constraints beschreiben sowohl Invarianten, die etwas über die Korrektheit des Applets aussagen als auch Vor- und Nachbedingungen für die manuell zu implementierenden Methoden. Es ist dann möglich, semi-automatisch eine formale Spezifikation zu erzeugen und basierend hierauf die OCL-Constraints sowie weitere Konsistenzchecks mit dem interaktiven Beweistool KeY [13] zu verifizieren. Im Gegensatz zu dem in dieser Arbeit vorgestellten An-

satz werden nur Korrektheits- und keine Sicherheitseigenschaften verifiziert. Außerdem wird nur das Java Card-Applet betrachtet, die Terminals bleiben außen vor. Während der Betrachtung der verschiedenen Fallstudien mit dem SecureMDD-Ansatz hat sich jedoch immer wieder gezeigt, dass es für die Sicherheit wichtig ist, die gesamte Anwendung, d.h. alle Kommunikationspartner und ihr Zusammenspiel zu betrachten und zu formalisieren und nicht nur einzelne Komponententypen. Aus diesem Grund wird im SecureMDD-Ansatz die komplette Anwendung modelliert (und verifiziert).

Weitere Ansätze für die Entwicklung sicherheitskritischer Anwendungen:
Es existiert eine Vielzahl weiterer Arbeiten, die sich mit der (modellgetriebenen) Entwicklung von sicheren Anwendungen beschäftigen. Anders als beim SecureMDD-Ansatz, der für Anwendungen, die auf kryptographischen Protokollen basieren, geeignet ist, haben diese Arbeiten jedoch einen anderen Fokus und werden deshalb nicht im Detail erläutert. Sie lassen sich unterteilen in Arbeiten zum Thema (rollenbasierte) Zugriffskontrolle (z.B. [3, 30], [11, 116], [108], [162]), Entwicklung von Service-Anwendungen (insbesondere Service-Orchestrierung und Compliance-Eigenschaften, z.B. [173], [120], [55],[29], [48], [124]) und webbasierten Systemen (z.B. [111]) sowie Requirements Engineering (z.B. [158]).

Der in dieser Arbeit vorgestellte Ansatz ermöglicht die Entwicklung einer Smart Card-Anwendung, bei der die auf abstrakter Ebene formal verifizierten Sicherheitseigenschaften auch für den automatisch generierten Quellcode der Anwendung gelten. Dies ist ein Alleinstellungsmerkmal, das keines der hier genannten Arbeiten erfüllt. Abgesehen von der im SecureMDD-Ansatz geltenden Verfeinerungsbeziehung zwischen dem generierten Code sowie dem formalen Modell, gibt es auch keine andere Arbeit, die vollständigen (d.h. nicht von Hand zu ergänzenden), lauffähigen Quellcode für eine Java Card-Anwendung generieren kann.

Teil II.

Modellierung einer sicherheitskritischen Chipkartenanwendung

4 Plattformunabhängige Modellierung einer Anwendung mit UML

Zusammenfassung: Der SecureMDD-Ansatz unterstützt die vollständige Modellierung einer sicherheitskritischen Anwendung mit UML. In diesem Kapitel wird die plattformunabhängige Modellierung detailliert erläutert. Damit aus dem Modell automatisch lauffähiger Quellcode und eine formale Spezifikation generiert werden können, muss die Modellierung dabei bestimmten Richtlinien folgen. Auch diese sind in diesem Kapitel beschrieben. Um zu garantieren, dass der generierte Code eine Verfeinerung des generierten formalen Modells ist, gibt es außerdem bestimmte Einschränkungen, die bei der Modellierung eingehalten werden müssen und vor der Ausführung der Transformationen überprüft werden. Auf diese wird ebenfalls eingegangen. Ziel des Kapitels ist es, einem Entwickler alle nötigen Informationen zu geben, die dieser für die Erstellung des plattformunabhängigen UML-Modells einer Anwendung benötigt. Die plattformunabhängige Modellierung ist in [130] sowie ausführlicher in [134] publiziert.

In diesem Kapitel sind zunächst, in Abschnitt 4.1, die für die Modellierung benötigten Grundlagen erläutert. Anschließend wird in Abschnitt 4.2 die statische Modellierung einer Anwendung mit Klassen- und Deploymentdiagrammen allgemein und in Abschnitt 4.3 für die Kopierkartenanwendung beschrieben. Abschnitt 4.4 erläutert die Modellierung der dynamischen Aspekte einer Anwendung mit Sequenz- und Deploymentdiagrammen allgemein, Abschnitt 4.5 enthält die dynamische Modellierung der Kopierkartenanwendung. Abschnitt 4.6 setzt die in diesem Kapitel vorgestellte Modellierung in den Kontext anderer Forschungsarbeiten.

4.1. Grundlagen der Modellierung

Für die Modellierung einer sicherheitskritischen Anwendung werden Datentypen wie zum Beispiel Nonces, Geheimnisse und kryptographische Schlüssel benötigt. Da diese Datentypen für jede Anwendung gleich sind, wurden sie zusammengefasst und stehen als Paket jedem Anwendungsmodell zur Verfügung. Weiterhin wird die Unified Modeling Language an die Domäne der sicherheitskritischen Anwendungen angepasst. Hierfür wurde die UML um ein Profil erweitert und eine Reihe von Stereotypen und Tagged Values definiert, um solche Anwendungen

modellieren zu können. In diesem Abschnitt werden die für die plattformunabhängige Modellierung benötigten Grundlagen erläutert.

Der Abschnitt gliedert sich wie folgt. Zunächst werden in Abschnitt 4.1.1 einige Begriffe definiert, die in dieser Arbeit verwendet werden. Anschließend werden in Abschnitt 4.1.2 die in SecureMDD vordefinierten Sicherheitsdatentypen beschrieben. Das SecureMDD UML-Profil ist in Abschnitt 4.1.3 beschrieben. Abschnitt 4.1.4 gibt einen Überblick über die für die plattformunabhängige Modellierung verwendeten UML-Diagrammtypen und beschreibt die Abhängigkeiten zwischen ihnen.

Der Vollständigkeit halber ist zu erwähnen, dass die domänenspezifische Sprache MEL ebenfalls eine Erweiterung der UML ist. Da die Sprache jedoch eng an die Modellierung mit Aktivitätsdiagrammen gekoppelt ist, wird in Kapitel 5, im Anschluss an die plattformunabhängige Modellierung, die Syntax und Semantik von MEL erläutert.

4.1.1. Grundlegende Begriffe

In diesem Abschnitt werden einige Begriffe erläutert, die in dieser Arbeit verwendet werden:

Definition:
Ein **Protokoll** ist eine Vereinbarung zwischen verschiedenen Kommunikationspartnern über den Inhalt und das Format der Nachrichten, die zwischen den Partnern ausgetauscht werden sowie die Reihenfolge, in der dies geschieht. In dieser Arbeit bezieht sich der Begriff Protokoll nicht nur auf den Austausch der Nachrichten, sondern auch auf die Verarbeitung einer empfangenen Nachricht durch einen Kommunikationspartner. Eine Protokollbeschreibung ist somit nur vollständig, wenn auch die Verarbeitung der Nachrichten sowie daraus resultierende Zustandsänderungen der Kommunikationspartner beschrieben sind.

Definition:
Ein **kryptographisches Protokoll** ist ein Protokoll, das durch die Verwendung kryptographischer Primitive und Algorithmen (wie z.B. Hashfunktionen, Zufallszahlengeneratoren, Verschlüsselungsfunktionen,..) bestimmte Schutzziele sicherstellt.

Definition:
Ein **Protokollschritt** ist ein Schritt eines Protokolls, der von einem Kommunikationspartner durchgeführt wird. Der erste Schritt eines Protokolls beginnt mit dem Start des Protokolls und endet mit dem Senden einer Nachricht an einen anderen Kommunikationspartner. Alle folgenden Protokollschritte beginnen mit dem Empfang einer Nachricht und enden mit dem Senden einer Nachricht an einen anderen Kommunikationspartner. Ein Protokollschritt beschreibt somit den Empfang einer Nachricht, dessen Verarbeitung auf Seiten des Empfängers, einer (optionalen) Änderung des Zustands des Empfängers sowie dem Senden einer weiteren Nachricht. Der letzte Schritt eines Protokolls beginnt mit dem Empfang einer Nachricht und endet mit dem Ende des Protokolls. Alternativ kann ein Protokollschritt anstatt mit dem Senden einer Nachricht auch mit einem Abbruch des Protokollschritts bei Auftreten eines Fehlers enden. In dieser Arbeit wird jeder Protokollschritt als atomar angesehen. Ein Angreifer kann durch Unterdrücken einer Nachricht zwar verhindern, dass ein Protokollschritt ausgeführt wird, kann jedoch nicht bewirken, dass Protokollschritte nur partiell durchgeführt werden.

Definition:
Als **Terminal** wird ein Gerät bezeichnet, das mindestens ein Smart Card-Lesegerät besitzt, über welches es mit einer Chipkarte kommunizieren kann. Im SecureMDD-Ansatz müssen die Lesegeräte für die Karten nicht explizit modelliert werden. Stattdessen kann ein Terminal in der Modellierung direkt mit einer Karte kommunizieren. Ein Terminal besitzt außerdem eine Ein- und Ausgabeschnittstelle für die Kommunikation mit einem Benutzer. Dies ist zum Beispiel eine graphische Benutzeroberfläche, über die der Nutzer der Anwendung Eingaben an das System machen kann und Ausgaben visualisiert werden können. Beispiele für ein Terminal sind ein Heim-PC mit angeschlossenem Kartenleser oder ein Ticketautomat, in den man eine Smart Card stecken und auf diese ein gekauftes Ticket laden kann.

Definition:
Als **Komponenten** einer Anwendung werden in dieser Arbeit alle Terminals und Smart Cards, die an der Anwendung beteiligt sind, bezeichnet.

4.1.2. Vordefinierte Sicherheitsdatentypen

Um bei der Modellierung nicht auf die Verwendung der UML-Datentypen angewiesen zu sein und darüber hinaus kryptographische Daten modellieren zu können, sind die bei der Modellierung zu verwendenden Basisdatentypen vordefiniert. Sie sind in Abbildung 4.1 als UML-Klassendiagramm dargestellt und werden in diesem Abschnitt erläutert.

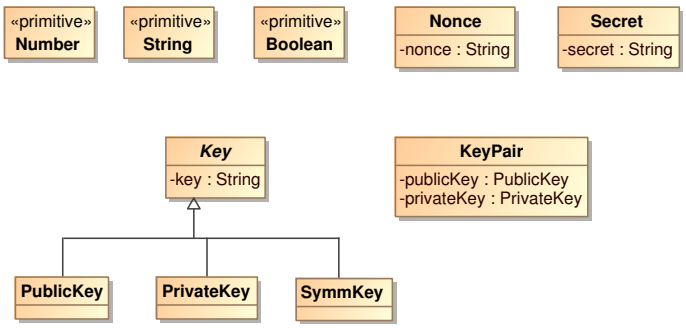

Abbildung 4.1.: Die primitiven Datentypen sowie Datentypen für kryptographische Schlüssel, Nonces und Secrets

Es gibt drei primitive Datentypen: `String` für die Repräsentation von Zeichenketten, `Boolean` für Wahrheitswerte und `Number` für (ganze) Zahlenwerte. Ein String darf nur aus Zeichen bestehen, für die es eine ASCII-Repräsentation gibt (d.h. das lateinische Alphabet in Groß- und Kleinbuchstaben, die zehn arabischen Ziffern sowie einige Satzzeichen). Umlaute dürfen nicht verwendet werden. Die Länge der verwendeten Strings ist begrenzt, die Längenangabe ist jedoch konfigurierbar.

Weiterhin gibt es den Datentyp `Nonce`. Die Klasse `Nonce` besitzt ein Attribut `nonce` vom Typ `String`, das den eigentlichen Zufallswert speichert.

Der Datentyp `Secret` repräsentiert ein Geheimnis, beispielsweise ein Passwort oder eine PIN, die der Benutzer für sich behalten sollte. Auch auf den Karten gespeicherte Werte, die zum

Beispiel der Authentifizierung dienen und dem Angreifer nicht bekannt sein dürfen, werden als `Secret` gespeichert. Die Klasse `Secret` besitzt ein Attribut `secret` vom Typ `String`, das das eigentliche Geheimnis speichert.

Darüber hinaus ist der abstrakte Typ `Key` für *kryptographische Schlüssel* definiert. Die zugehörige Klasse hat ein Attribut `key` vom Typ `String`, das den eigentlichen Schlüssel speichert. Von der Klasse abgeleitet sind die konkreten Klassen `PublicKey`, `SymmKey` und `PrivateKey`, die einen öffentlichen, symmetrischen bzw. privaten Schlüssel repräsentieren. Bei der Verschlüsselung werden die symmetrische und die asymmetrische Verschlüsselung unterschieden. Bei der *symmetrischen Verschlüsselung* verwenden beide Teilnehmer den gleichen Schlüssel, dieser muss geheim bleiben. Bei der *asymmetrischen Verschlüsselung* wird ein Schlüsselpaar bestehend aus einem öffentlichen und einem privaten Schlüssel verwendet. Ein Schlüsselpaar ist dabei einem Teilnehmer zugeordnet. Soll ein Dokument verschlüsselt werden, geschieht dies mit dem öffentlichen Schlüssel. Entschlüsseln kann das Dokument nur der Besitzer des privaten Schlüssels. Dieser muss geheim bleiben.

Um ein *asymmetrisches Schlüsselpaar* zu repräsentieren, gibt es den Datentyp `KeyPair`. Die Klasse `KeyPair` besitzt ein Attribut `publicKey` vom Typ `PublicKey`, das den öffentlichen Schlüssel des Schlüsselpaares speichert. Das Attribut `privateKey` vom Typ `PrivateKey` speichert den privaten Schlüssel des Paares. Die Klasse KeyPair wird nur für die Verwendung auf Terminalseite unterstützt. Die Klasse dient lediglich dem Erzeugen von Schlüsselpaaren. Da dies bei der Initialisierung der Komponenten und außerhalb der Karte auf Terminalseite geschieht, reicht es aus, die Klasse KeyPair nur für Terminals zu unterstützen.

Die folgenden Datentypen für verschlüsselte Daten, Signaturen, Hash- und MAC-Werte werden nur in den Aktivitätsdiagrammen bei der Deklaration von lokalen Variablen mit der domänenspezifischen Sprache MEL verwendet (siehe Abschnitt 4.4.2.2). Der Vollständigkeit halber sind sie jedoch ebenfalls in den Sicherheitsdatentypen aufgeführt.

Die Datentypen sind in Abbildung 4.2 als UML-Klassen dargestellt.

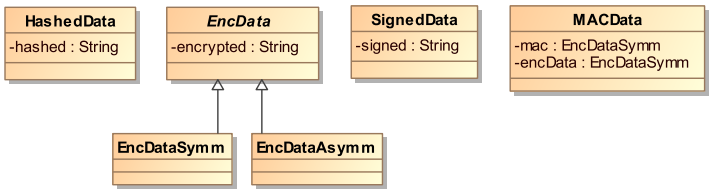

Abbildung 4.2.: Datentypen für verschlüsselte Dokumente, Signaturen, Hashwerte und MAC-Werte

Die gemeinsame abstrakte Oberklasse `EncData` repräsentiert ein verschlüsseltes Datum im Allgemeinen. Dieses ist im Attribut `encrypted` vom Typ `String` gespeichert. Von dieser Klasse abgeleitet sind die konkreten Klassen `EncDataSymm` zur Repräsentation symmetrisch verschlüsselter Daten sowie die Klasse `EncDataAsymm` für asymmetrisch verschlüsselte Daten. Signierte Daten werden durch die Klasse `SignedData` repräsentiert. Dieses besitzt das Attribut `signed`, das das signierte Datum speichert. Der Hashwert eines Datums wird durch die Klasse `HashedData` repräsentiert. Diese besitzt ein Attribut `hashed` vom Typ `String`, das den eigentlichen Hashwert speichert. Bildet man über einem Objekt `d` einen MAC-Wert, ist das Ergebnis ein Objekt vom Typ `MACData`. Dieses hat zwei Attribute. Das Attribut

Stereotyp	Erweiterte Metaklasse	Kurzbeschreibung
Smartcard	Class	annotiert eine Klasse, die eine Smart Card-Komponente repräsentiert
Terminal	Class	annotiert eine Klasse, die eine Terminal-Komponente repräsentiert
User	Class	annotiert eine Klasse, die einen Benutzer repräsentiert
Message	Class	annotiert eine Klasse, die eine Nachricht repräsentiert
Usermessage	Class	annotiert eine Klasse, die eine Benutzernachricht repräsentiert

Tabelle 4.1.: Stereotypen für die Komponenten- und Nachrichtentypen

`encrypted` speichert das (mit einem symmetrischen Schlüssel `cryptkey`) verschlüsselte Objekt `d`. Das Attribut `mac` speichert den (mit einem symmetrischen Schlüssel `mackey`) verschlüsselten Hashwert des Objekts `d`. Dieser dient der Integrität und stellt sicher, dass das im Attribut `mac` gespeicherte Objekt (das durch Hashen und anschließender Verschlüsselung des Objekts `d` erzeugt wurde) von einer Instanz, die den symmetrischen Schlüssel `mackey` kennt, erzeugt wurde. Um ein Objekt vom Typ `MACData` zu erzeugen, werden zwei symmetrische Schlüssel benötigt: Der Schlüssel `cryptkey` zum Verschlüsseln des Dokuments sowie der Schlüssel `mackey` zum Bilden des eigentlichen Message Authentication Codes. Die Klasse `MACData` repräsentiert einen MAC-Wert und besitzt zwei Attribute. Das Attribut `encrypted` speichert das symmetrische verschlüsselte Klartextobjekt, das Attribut `mac` enthält den eigentlichen MAC-Wert.

4.1.3. UML-Profil für sicherheitskritische Anwendungen

Das für SecureMDD definierte UML-Profil lässt sich in fünf Teile untergliedern:

- Stereotypen für die Beschreibung der Komponenten- und Nachrichtentypen
- Stereotypen für die Modellierung von kryptographischen Daten
- eine Sammlung von Stereotypen für die Protokollmodellierung
- Stereotypen für die Definition des Angreifers
- Stereotypen zur Strukturierung eines UML Projekts

Im Folgenden wird ein Überblick über das UML-Profil gegeben. Die Verwendung ist in den Abschnitten 4.2 bis 4.5 ausführlich beschrieben.

Stereotypen für die Beschreibung der Komponenten- und Nachrichtentypen

In Tabelle 4.1 sind die Stereotypen für die Komponenten- und Nachrichtentypen angegeben.

Die Komponenten der zu modellierenden Anwendung, also die beteiligten Smart Cards und Terminals, sowie die Benutzer des Systems, sind als Klassen im Klassendiagramm modelliert und werden mit Stereotypen versehen. Für die Smart Cards ist der Stereotyp ≪Smartcard≫ und für die Terminals ist der Stereotyp ≪Terminal≫ vorgesehen. Da die Komponenten durch UML-Klassen beschrieben werden, erweitern diese beiden Stereotypen die Metaklasse `Class`. Die Benutzer der Anwendung werden ebenfalls durch UML-Klassen definiert. Diese tragen den Stereotyp ≪User≫, der ebenfalls von der Metaklasse `Class` abgeleitet ist.

Die Nachrichten (-typen) werden als Datentypen in einem Klassendiagramm definiert. Jeder Nachrichtentyp ist durch eine eigene Klasse beschrieben. Wir unterscheiden dabei zwei Arten von Nachrichten. Zum einen sind dies die Nachrichten, die zwischen einer Smart Card und einem Terminal ausgetauscht werden. Diese werden `Messages` genannt. Zum anderen gibt es die Nachrichten, die vom Benutzer des Systems an dieses sowie vom System an den Benutzer geschickt werden. Diese werden `Usermessages` genannt. Um Nachrichtenklassen als solche erkennbar zu machen, gibt es die Stereotypen ≪Message≫ und ≪Usermessage≫, die ebenfalls die Metaklasse `Class` erweitern.

Stereotypen für die Beschreibung von kryptographischen Daten

Weitere Stereotypen ermöglichen die Darstellung von signierten, verschlüsselten und gehashten Daten sowie des MAC-Wertes und von Zertifikaten in Klassendiagrammen. Diese sind in Tabelle 4.2 dargestellt.

Digitale Signaturen bieten die Möglichkeit, zu einem Datum eine eindeutige Zeichenfolge zu berechnen. Mit dessen Hilfe lässt sich von einer dritten Person überprüfen, dass das Datum seit Berechnung der Signatur nicht verändert wurde. Weiterhin ist es möglich, mittels der Signatur den Urheber des Datums zu authentifizieren. Es gibt mehrere Möglichkeiten digitale Signaturen zu realisieren. SecureMDD verwendet digitale Signaturen basierend auf RSA. Ein Teilnehmer A signiert ein Dokument mit seinem privaten Schlüssel. Jeder andere Teilnehmer, der den öffentlichen Schlüssel von A kennt, ist nun in der Lage nachzuweisen, dass A das Dokument signiert hat. Hierfür wird mithilfe des öffentlichen Schlüssels die Signatur verifiziert.

Ein *Message Authentication Code* (MAC) ermöglicht es, die Integrität eines Datums zu überprüfen. Derjenige, der das Datum empfängt, kann mithilfe des MAC-Wertes prüfen, ob das Datum tatsächlich von der erwarteten Instanz erzeugt und seit Erzeugung des MAC-Wertes auch nicht mehr verändert wurde. Message Authentication Codes verwenden Verschlüsselung sowie Hashwerte. In SecureMDD wird das Datum, dessen Integrität sichergestellt werden soll, zusätzlich verschlüsselt, so dass ein Angreifer dieses Datum nicht lesen kann.

Klartextdaten, die während eines Protokolllaufs verschlüsselt, signiert oder gehasht werden sollen, werden als eigene Klassen definiert und mit entsprechenden Stereotypen gekennzeichnet. Daten, die im Laufe eines Protokolllaufs signiert werden, werden mit Stereotyp ≪SignData≫ annotiert. Daten, die (symmetrisch oder asymmetrisch) verschlüsselt oder über denen ein MAC-Wert gebildet werden soll, tragen den Stereotyp ≪PlainData≫. Daten, die gehasht werden, sind mit Stereotyp ≪HashData≫ annotiert. Alle drei Stereotypen erweitern die Metaklasse `Class`.

Dass ein Datum in signierter Form gespeichert wird, ist durch den Stereotyp ≪signed≫ gekennzeichnet. Dieser Stereotyp annotiert das Assoziationsende, dessen Assoziation auf das

Stereotyp	Erweiterte Metaklasse	Kurzbeschreibung
SignData	Class	annotiert eine Datenklasse, die während eines Protokolllaufs signiert wird
signed	Property	annotiert ein Attribut oder Assoziationsende, das eine Signatur speichert
PlainData	Class	annotiert eine Datenklasse, die im Protokolllauf verschlüsselt oder über der ein MAC-Wert gebildet wird
encrypted	Property	annotiert ein Attribut oder Assoziationsende, das ein symmetrisch verschlüsseltes Datum speichert
encryptedAsymm	Property	annotiert ein Attribut oder Assoziationsende, das ein asymmetrisch verschlüsseltes Datum speichert
HashData	Class	annotiert eine Datenklasse, die während eines Protokolllaufs gehasht wird
hashed	Property	annotiert ein Attribut oder Assoziationsende, das einen Hashwert speichert
MAC	Property	annotiert ein Attribut oder Assoziationsende, das einen MAC-Wert speichert
Certificate	Class	annotiert eine Klasse, die ein Zertifikat repräsentiert

Tabelle 4.2.: Stereotypen für kryptographische Daten

entsprechende Klartextdatum (das mit ≪SignData≫ gekennzeichnet ist) zeigt. Der Stereotyp ≪signed≫ kann für Assoziationsenden (und Attribute) verwendet werden und erweitert die Metaklasse `Property`.

Dass ein Datum in verschlüsselter Form gespeichert ist, wird durch den Stereotyp ≪encrypted≫ (wenn es symmetrisch verschlüsselt ist) bzw. ≪encryptedAsymm≫ (wenn es asymmetrisch verschlüsselt ist) gekennzeichnet. Wie auch beim Signieren von Daten annotieren die Stereotypen ≪encrypted≫ und ≪encryptedAsymm≫ das Assoziationsende, das auf das zu verschlüsselnde Datum zeigt. Die Klasse, die den Klartext repräsentiert, ist mit dem Stereotyp ≪PlainData≫ annotiert.

Das Bilden eines Hashwertes wird analog modelliert. Der Stereotyp ≪hashed≫ gibt an, dass ein Wert in gehashter Form gespeichert wird. Er annotiert ein Assoziationsende und erweitert die Metaklasse `Property`. Die Klasse, die die Klartextdaten enthält, trägt den Stereotyp ≪HashData≫.

Äquivalent zur Verschlüsselung wird das Bilden eines MAC-Wertes modelliert. Der MAC-Wert wird durch Verschlüsseln und Signieren eines Objekts gebildet. Die Definition eines eigenen Stereotyps hierfür ist also nicht zwingend notwendig, sondern das Bilden des MAC-Wertes lässt sich indirekt auch mithilfe der Stereotypen ≪encrypted≫ und ≪signed≫ ausdrücken. Um die Modellierung übersichtlicher zu machen, wurde jedoch zusätzlich der Stereotyp ≪MAC≫ eingeführt. Die Stereotypen ≪encrypted≫, ≪encryptedAsymm≫ und ≪MAC≫ annotieren Assoziationsenden (und Attribute) und erweitern die Metaklasse `Property`.

Öffentliche Schlüssel sind nicht geheim und somit für viele Instanzen bzw. Personen zugänglich. Um öffentliche Schlüssel zu verteilen werden *digitale Zertifikate* verwendet. Die Idee hierbei ist, dass ein öffentlicher Schlüssel zusammen mit dem Namen seines Besitzers sowie evtl. weiteren Informationen von einer vertrauensvollen Zertifizierungsstelle signiert wird. Dieser Stelle vertrauen sowohl der Besitzer des öffentlichen Schlüssels als auch die Instanzen, die den öffentlichen Schlüssel verwenden möchten. Die Zertifizierungsstelle bestätigt durch ihre Signatur die Authentizität des öffentlichen Schlüssels sowie der weiteren, im Zertifikat enthaltenen, Daten. Jede Instanz kann durch Überprüfen der Signatur (mit dem öffentlichen Schlüssel des Zertifikatherausgebers) verifizieren, dass das Zertifikat authentisch ist.

SecureMDD definiert den Stereotyp ≪Certificate≫, der die Metaklasse `Class` erweitert. Eine mit ≪Certificate≫ annotierte Klasse repräsentiert ein Zertifikat.

Stereotypen für die Protokollmodellierung

Für die Modellierung der Protokolle werden ebenfalls einige Stereotypen definiert. Diese sind in der Tabelle 4.3 abgebildet.

Anwendungen, die auf kryptographischen Protokollen basieren, benötigen meistens einen expliziten Zustand, um zu speichern in welchem Schritt innerhalb eines Protokolls sich eine Komponente gerade befindet bzw. welchen Nachrichtentyp die Komponente als nächstes erwartet. Die möglichen Zustände einer Komponente werden in einer Enumerationklasse definiert. Die Komponentenklasse muss eine Assoziation zu dieser Enumerationklasse besitzen, deren Ende mit dem Stereotyp ≪status≫ annotiert wird. Der Stereotyp ≪status≫ erweitert die Metaklasse `Property`.

Stereotyp	Erweiterte Metaklasse(n)	Kurzbeschreibung
status	Property	annotiert das Assoziationsende zu der Statusklasse einer Komponente
Constant	Class	annotiert eine Klasse, die Konstanten definiert
Manual	Class, Operation	annotiert eine Klasse mit manuell zu implementierenden Operationen bzw. annotiert manuelle Operationen
Initialize	Property	annotiert ein Attribut oder Assoziationsende, das während der Personalisierungsphase initialisiert werden muss

Tabelle 4.3.: Stereotypen für die Protokollmodellierung

Eine Klasse, die Konstanten definiert ist mit dem Stereotyp ≪Constant≫ annotiert, die Konstanten sind als Attribute der Klasse angegeben. Dieser Stereotyp erweitert die Metaklasse `Class`.

SecureMDD bietet die Möglichkeit, Signaturen für Methoden anzugeben, die später von Hand implementiert werden. Diese können im Klassendiagramm als Operationen definiert werden. Manuell zu implementierende Methoden können sowohl für Daten- als auch für Komponentenklassen (d.h. Klassen mit Stereotyp ≪Smartcard≫ oder ≪Terminal≫) angegeben werden und müssen mit Stereotyp ≪Manual≫ annotiert sein. In manchen Fällen kommt es vor, dass die Funktionalität einer Operation keiner Daten- oder Komponentenklasse zugeordnet werden kann. In diesen Fällen ist es möglich, eine Klasse zu definieren, die nur manuell zu implementierende Methoden enthält. Diese Klasse muss ebenfalls im Klassendiagramm modelliert werden und mit Stereotyp ≪Manual≫ annotiert sein. Der Stereotyp ≪Manual≫ erweitert die Metaklassen `Class` und `Operation`.

Vor Inbetriebnahme einer Anwendung müssen die Smart Cards und Terminals in der Regel initialisiert werden. Für Smart Cards ist hierfür eine Personalisierungsphase vorgesehen, in der zum Beispiel kryptographische Schlüssel und Geheimnisse wie eine PIN-Nummer gesetzt werden. Ebenso müssen auch einige Datenfelder der Terminals schon vor Beginn des Produktivbetriebs gesetzt werden. SecureMDD stellt einen Mechanismus bereit, der die Initialisierung der modellierten Smart Card- und Terminalkomponenten unterstützt. Bei der Modellierung einer Anwendung mit UML muss bereits angegeben werden, welche Attribute und Assoziationsenden einer Komponentenklasse initialisiert werden sollen. Hierfür wird der Stereotyp ≪Initialize≫ annotiert. Dieser Stereotyp erweitert die Metaklasse `Property`.

Stereotypen für Sicherheitseigenschaften und den Angreifer

Für Attribute und Assoziationsenden, die nur für die Formulierung von Sicherheitseigenschaften verwendet werden, sowie für die Definition der Angreiferfähigkeiten wird ebenfalls jeweils ein Stereotyp definiert. Diese sind in Tabelle 4.4 dargestellt.

Stereotyp	Erweiterte Metaklasse	Kurzbeschreibung
Ghostvariable	Property	annotiert eine Property, die für die Formalisierung der Sicherheitseigenschaften benötigt wird.
Threat	CommunicationPath (mit den Tags read, send, suppress)	beschreibt die Angreiferfähigkeiten

Tabelle 4.4.: Stereotypen für Sicherheitseigenschaften und den Angreifer

Für die Definition der Sicherheitseigenschaften ist es manchmal notwendig Attribute oder Assoziationen (bzw. Assoziationsenden) zu definieren, die für die eigentliche Funktionalität der Anwendung nicht benötigt werden. Sie haben einzig den Zweck, die betrachtete Sicherheitseigenschaft ausdrücken bzw. formalisieren zu können und werden nur bei der Generierung des formalen Modells berücksichtigt. Diese Attribute und Assoziationsenden werden mit dem Stereotyp ≪Ghostvariable≫ annotiert. Dieser Stereotyp erweitert die Metaklasse `Property`.

Deploymentdiagramme werden für die Modellierung der Kommunikationsinfrastruktur sowie die Definition des Angreifers verwendet. Der Angreifer hat die Fähigkeit, die Kommunikationskanäle zwischen den Komponenten abzuhören, Nachrichten über diese Kanäle zu versenden oder Nachrichten zu unterdrücken. Um diese Fähigkeiten modellieren zu können, wird der Stereotyp ≪Threat≫ verwendet. Dieser erweitert das Metamodellelement `CommunicationPath`, das für die Beschreibung eines zwischen zwei Komponenten bestehenden Kommunikationskanals verwendet wird. Der Stereotyp besteht aus drei booleschen Tags, die angeben, ob die Verbindung vom Angreifer abgehört werden kann (Tag `read`), Nachrichten vom Angreifer über diese Verbindung gesendet werden können (Tag `send`) oder Nachrichten, die über diese Verbindung geschickt werden, vom Angreifer unterdrückt werden können (Tag `suppress`).

Stereotypen zur Strukturierung eines Projekts

Um das UML-Projekt übersichtlicher zu gestalten, müssen die einzelnen UML-Diagrammtypen innerhalb des Projekts strukturiert werden. Aus diesem Grund besteht die Projektvorlage aus fünf verschiedenen Paketen, die später die verschiedenen Diagrammtypen enthalten. Dies sind Klassendiagramme, Sequenzdiagrammme, Aktivitätsdiagramme und Deploymentdiagramme sowie ein Paket, das die vordefinierten Sicherheitsdatentypen (siehe 4.1.2) enthält. Die für die Strukturierung definierten Stereotypen sind in Tabelle 4.5 dargestellt.

Um bei den Modelltransformationen einfacher auf diese Informationen zugreifen zu können, sind die Pakete entsprechend mit den Stereotypen ≪ClassDiagram≫, ≪SequenceDiagram≫, ≪ActivityDiagram≫, ≪DeploymentDiagram≫ und ≪SecurityDatatypes≫ annotiert. Diese Stereotypen erweitern die UML-Metaklasse `Package`.

Stereotyp	Erweiterte Metaklasse	Kurzbeschreibung
ClassDiagram	Package	annotiert das Klassendiagramm-Package
SequenceDiagram	Package	annotiert das Sequenzdiagramm-Package
ActivityDiagram	Package	annotiert das Aktivitätsdiagramm-Package
DeploymentDiagram	Package	annotiert das Deploymentdiagramm-Package
SecurityDatatypes	Package	annotiert das Paket mit den Sicherheitsdatentypen

Tabelle 4.5.: Stereotypen für die Strukturierung eines Projekts

4.1.4. Verwendete Diagrammtypen und deren Abhängigkeiten

Das plattformunabhängige Modell einer Anwendung lässt sich in zwei Teile untergliedern: einen statischen sowie einen dynamischen Teil. Einen Überblick über die für die Modellierung verwendeten Diagrammtypen sowie die Abhängigkeiten zwischen ihnen gibt Abbildung 4.3.

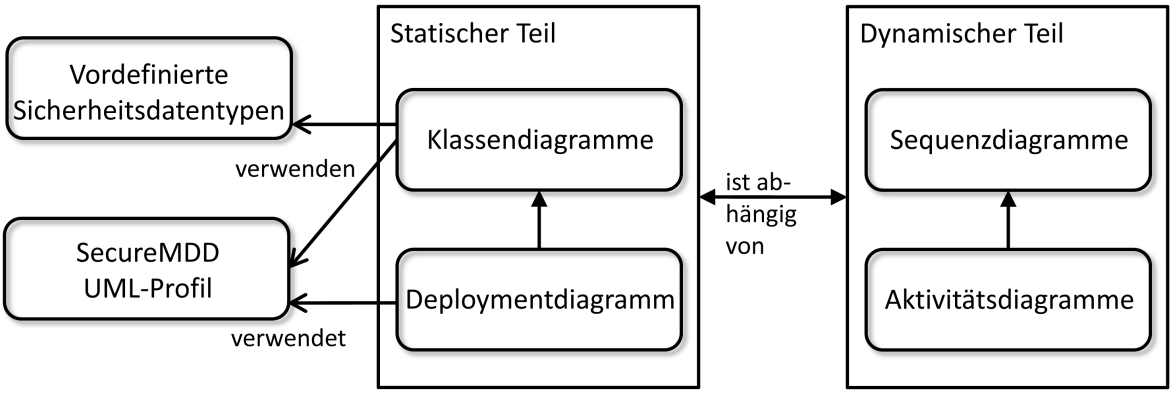

Abbildung 4.3.: Überblick über die verwendeten UML-Diagrammtypen

Für die Modellierung der statischen Aspekte werden Klassen- und Deploymentdiagramme verwendet. Die Klassendiagramme beschreiben die an der Anwendung beteiligten Komponenten, die Benutzer der Anwendung sowie die verwendeten Daten- und Nachrichtentypen und verwenden die in Abschnitt 4.1.2 vorgestellten Sicherheitsdatentypen. Um die Kommunikationsinfrastrukutur und die Fähigkeiten des Angreifers zu beschreiben, werden Deploymentdiagramme verwendet. Sowohl die Klassen- als auch die Deploymentdiagramme benutzen das in Abschnitt 4.1.3 vorgestellte UML-Profil.

Die dynamische Sicht des zu entwickelnden Systems wird mittels Sequenz- und Aktivitätsdiagrammen definiert. Die Sequenzdiagramme beinhalten eine grobe Beschreibung der Protokolle

der Anwendung. Angelehnt an die klassische Notation für kryptographische Protokolle sind hier die an der Kommunikation beteiligten Komponenten sowie die zwischen ihnen ausgetauschten Nachrichten modelliert. Das Verhalten der Anwendung bei Auftreten eines Fehlers, die eigentliche Verarbeitung der Nachrichten und der daraus resultierenden Änderung des Zustands einer Komponente sind nicht dargestellt. Zur detaillierten Modellierung der dynamischen Sicht der Anwendung werden UML-Aktivitätsdiagramme verwendet. Diese sind eine Verfeinerung der Sequenzdiagramme, da sie das komplette dynamische Verhalten des Systems, einschließlich des Austauschs der Nachrichten, beinhalten.

Da zwischen den UML-Diagrammen gewisse Abhängigkeiten bestehen, können sie nicht unabhängig voneinander entwickelt werden. Die Deploymentdiagramme beschreiben die Kommunikation zwischen den in den Klassendiagrammen definierten Komponenten sowie den Benutzern der Anwendung. Sie sind somit abhängig von den Klassendiagrammen. Die Aktivitätsdiagramme sind eine Verfeinerung der Sequenzdiagramme und müssen sich an die in den Sequenzdiagrammen vorgegebene Kommunikation halten. Die Aktivitäts- und Sequenzdiagramme stehen somit ebenfalls in einer Abhängigkeitsbeziehung. Zwischen dem statischen und dem dynamischen Teil bestehen ebenfalls enge Abhängigkeitsbeziehungen. Die Klassendiagramme definieren die in den Sequenz- und Aktivitätsdiagrammen verwendeten Nachrichtentypen. Die Deploymentdiagramme geben die für den dynamischen Teil wichtige Kommunikationsstruktur vor. Weiterhin werden in den Aktivitätsdiagrammen die Zustände und Daten einer Komponente verändert. Diese sind ebenfalls in den Klassendiagrammen definiert.

4.2. Statische Modellierung

In diesem Abschnitt wird die plattformunabhängige Modellierung der statischen Aspekte einer sicherheitskritischen Anwendung vorgestellt. Das Hauptaugenmerk liegt dabei auf dem Verständnis der Modellierung einer Anwendung. Ziel dieser und der folgenden Abschnitte ist es, dass der Leser anschließend in der Lage ist, selbstständig eine Anwendung mit dem Ansatz zu modellieren. Für die Generierung von Quellcode sowie der formalen Spezifikation aus dem Modell ist es wichtig, dass dabei die vorgegebenen Modellierungsrichtlinien eingehalten werden. Weiterhin gelten für die Modellierung bestimmte Einschränkungen (z.B. die Zyklenfreiheit der Klassen im Klassendiagramm oder nur eine sehr eingeschränkte Verwendung von Vererbung). Diese Einschränkungen sind notwendig, um zu garantieren, dass der generierte Code eine Verfeinerung des formalen Modells ist und somit die Sicherheitseigenschaften auch für den Code gelten (siehe Kapitel 10). Die Modellierungsrichtlinien sowie die erforderlichen Einschränkungen an die Modellierung sind in diesem Abschnitt (sowie dem Abschnitt über die dynamische Modellierung) ebenfalls ausführlich erläutert. Ihre Einhaltung wird vor der Ausführung der Modelltransformation durch eine automatische Validierung des Modells überprüft. Erst wenn diese Validierung erfolgreich ist, ist das Modell gültig und lässt sich in Code bzw. die formale Spezifikation transformieren.

In Abschnitt 4.2.1 ist die Modellierung mit Klassendiagrammen beschrieben. Abschnitt 4.2.2 stellt die Modellierung der Kommunikationsstruktur und des Angreifers mit Deploymentdiagrammen vor.

4.2.1. Klassendiagramme

4.2.1.1. Modellierung der Komponenten

Jede Komponente der zu entwickelnden Anwendung wird als eigene Klasse im Klassendiagramm modelliert. Durch Annotation des Stereotyps ≪Smartcard≫ bzw. ≪Terminal≫ wird angegeben, um welchen Komponententyp es sich handelt. Neben den Karten- und den Terminalkomponten gibt es außerdem noch die Benutzer, die mit dem System kommunizieren und so Vorgänge anstoßen und am Ende eines Vorgangs über dessen Ergebnis informiert werden möchten. Die Benutzer kommunizieren mit den Terminals der Anwendung. Die Terminals verfügen über eine Eingabemöglichkeit, in der Regel einer grafischen Benutzerschnittstelle, und können so auch Ausgaben des Systems darstellen. Eine direkte Kommunikation eines Benutzers mit der Karte ist nicht möglich. Jeder Benutzer wird ebenfalls durch eine eigene Klasse modelliert, die mit dem Stereotyp ≪User≫ annotiert ist.

In einigen Anwendungen kommt es vor, dass es verschiedene Arten eines Komponententyps gibt (z.B. verschiedene Arten von Terminals), diese aber gewisse Gemeinsamkeiten haben. In diesem Fall bietet es sich an, eine Oberklasse zu definieren und die verschiedenen Komponenten jeweils als Subklasse davon zu definieren. In diesem Fall muss die Oberklasse abstrakt sein und den Stereotyp zur Angabe der Komponente tragen. Ein gültiges UML-Modell muss mindestens jeweils eine Smart Card-, Terminal- und Userklasse besitzen. Mehrfachvererbung ist nicht erlaubt. Auch Vererbung über mehrere Klassen (C ist Subklasse von B, B ist Subklasse von A) ist nicht erlaubt.

In Abbildung 4.4 sind die Komponentenklassen `Karte`, `Benutzer` sowie die Terminals `konkretesTerminal1` und `konkretesTerminal2` modelliert. Die beiden Terminalklassen haben eine gemeinsame abstrakte (= der Name der Klasse ist kursiv geschrieben) Superklasse `Terminal`.

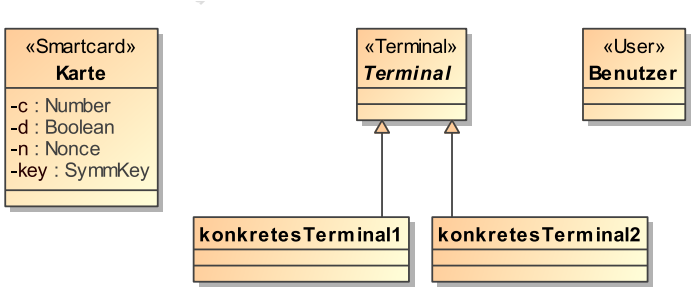

Abbildung 4.4.: Modellierung der Komponenten einer Anwendung

4.2.1.2. Modellierung der Daten einer Komponente

Die Komponentenklassen können Attribute besitzen. Der Typ jedes Attributs muss einem Datentyp aus den Sicherheitsdatentypen entsprechen. Welche Sichtbarkeit die Attribute (und auch die Assoziationsenden) haben, ist für die Modellierung mit SecureMDD nicht relevant.

Diese Information wird bei der Generierung der plattformspezifischen Modelle ignoriert und kann deshalb beliebig gewählt werden.

Außerdem kann eine Komponentenklasse Assoziationen zu weiteren Datenklassen besitzen. Datenklassen sind Klassen, die keine Komponentenklassen, keine Nachrichten- oder Statusklassen sowie keine Konstantenklassen oder Klassen mit Stereotyp ≪Manual≫ sind. Klassen mit Stereotyp `PlainData`, `HashData` und `SignData` sind ebenfalls Datenklassen. Die Datenklassen können ebenfalls wieder Attribute und Assoziationen haben. Die Assoziationen müssen (einseitig) gerichtet sein und das gerichtete Ende muss auf eine Datenklasse zeigen. Das gerichtete Ende muss einen Namen sowie eine Multiplizität haben. Als Multiplizitäten sind Zahlen größer als null sowie Intervalle der Form 0..10 erlaubt. Assoziationsenden mit Multiplizität größer als eins werden als Listen interpretiert. Stern-Assoziationen oder die Intervallobergrenze `n` für die Definition einer Liste beliebiger Länge sind nur für Terminals erlaubt. Bei Smart Cards ist dies aufgrund des begrenzten Speicherplatzes nicht möglich.

Abbildung 4.5 zeigt die Smart Card-Komponente `Karte` mit einigen Attributen und Assoziationen zu Datenklassen.

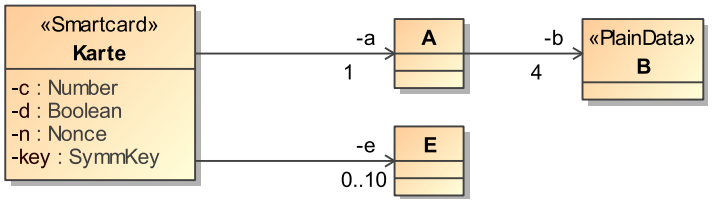

Abbildung 4.5.: Modellierung der Daten einer Anwendung

Die Klasse `Karte` besitzt vier Attribute, deren Typ einem Datentyp aus den Sicherheitsdatentypen entspricht. Die Karte hat außerdem eine Assoziation zu der Klasse `A`. Diese Assoziation muss gerichtet sein und das gerichtete Ende muss auf die Datenklasse `A` zeigen. Das gerichtetet Ende muss einen Namen (`a`) und eine Multiplizität (`1`) haben. Sind in den in dieser Arbeit abgedruckten UML-Modellen keine Multiplizitäten angegeben, besitzen die entsprechenden Assoziationsenden die Multiplizität eins. Die Klasse `A` hat wiederum eine Assoziation zur Datenklasse `B`, für die dieselben Regeln gelten. Als Multiplizität ist hier `4` angegeben, d.h. das Objekt `A` speichert eine Liste mit Elementen vom Typ `B`, die die Länge `4` und den Namen `b` hat. Außerdem besitzt die Klasse `Karte` eine Assoziation zur Klasse `E`, das Assoziationsende hat die Multiplizität `0..10`. Dies bedeutet, dass die Karte eine Liste speichert. Diese Liste enthält null bis zehn Objekte vom Typ `E`. Die Anzahl der gespeicherten Objekte kann je nach Protokolllauf unterschiedlich sein. Die Angabe einer Multiplizität größer als eins ist nur für Assoziationsenden möglich. Bei den Attributen einer Klasse werden Multiplizitäten größer als Eins ignoriert. Soll eine Liste von zum Beispiel Nonces modelliert werden, müssen diese in einer separaten Klasse gekapselt werden. Die Assoziation zu dieser Klasse besitzt dann eine Multiplizität größer als eins und wird somit als Liste behandelt.

Das UML-Modell muss zyklenfrei sein, d.h. insbesondere, dass Zyklen zwischen den Datenklassen nicht erlaubt sind. Es ist also nicht möglich, eine beidseitig gerichtete Assoziation anzugeben oder einen Zyklus innerhalb der Datenklassen zu erzeugen. Ein Klassendiagramm ist zyklenfrei, wenn es nicht möglich ist, von einer beliebigen Klasse ausgehend den gerich-

teten Assoziationen (über mehrere Klassen hinweg) zu folgen und die Ausgangsklasse wieder zu erreichen. Der Grund hierfür ist die benötigte Äquivalenz zwischen Code und formalem Modell.

Die Klassendiagramme dürfen außerdem keine Enumerations (außer für die Modellierung des Zustands einer Komponente) sowie Interfaces beinhalten. Auch Generalisierungen dürfen nur für Komponentenklassen verwendet werden, für Datenklassen sind sie nicht erlaubt.

4.2.1.3. Modellierung des Zustands einer Komponentenklasse

Eine Komponente muss sich während eines Protokolllaufs merken, an welcher Stelle im Protokoll sie sich gerade befindet. Dies ist notwendig, um beim Empfang einer Nachricht zu erkennen, ob diese vom Typ her zu der als nächstes erwarteten Nachricht passt. Auf diese Weise wird verhindert, dass ein Angreifer ein Protokoll an einer Stelle (durch Unterdrücken von Nachrichten) abbrechen oder ein Protokoll an beliebiger Stelle beginnen kann.

Der Zustand einer Komponente wird mittels einer Enumeration modelliert. Die möglichen Zustände der Komponente sind innerhalb dieser Klasse als Enumerationliterale angegeben. Die Komponentenklasse muss eine gerichtete Assoziation zu der Enumerationklasse besitzen, das Assoziationsende zeigt auf die Enumerationklasse. Das gerichtete Assoziationsende ist mit dem Stereotyp ≪status≫ annotiert und muss einen Namen haben.

Die möglichen Zustände (d.h. die Enumerationliterale) müssen eindeutig sein, d.h. es darf keine zwei Zustände mit demselben Namen geben. Auch die möglichen Zustände unterschiedlicher Komponenten (d.h. wenn sie in unterschiedlichen Enumerationklassen definiert sind) müssen unterschiedlich benannt sein. Jede Komponentenklasse muss genau eine gerichtete Assoziation zu einer Zustandsklasse besitzen. Besitzt eine Komponentenklasse noch eine Superklasse, kann der Zustand entweder für die Superklasse oder für die abgeleitete Klasse definiert werden. Assoziationen zwischen Daten- und Zustandsklassen sind nicht erlaubt.

In Abbildung 4.6 ist die Modellierung des Zustands einer Komponentenklasse dargestellt.

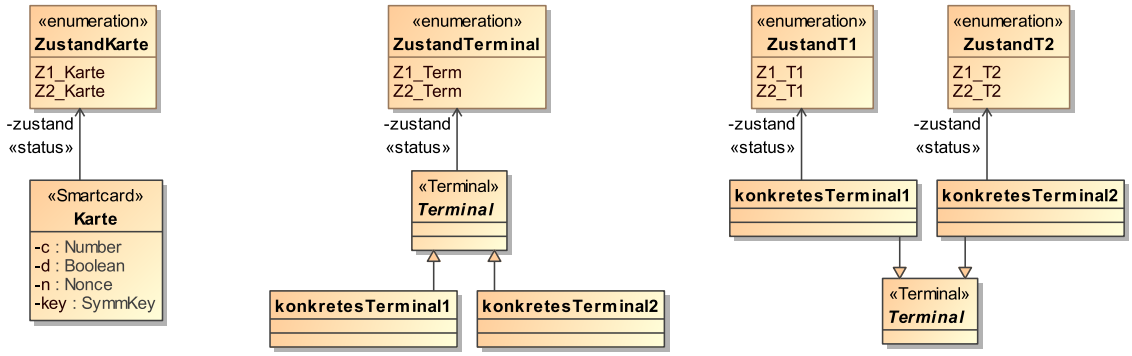

Abbildung 4.6.: Modellierung des Zustands einer Komponente

Der Zustand der Smart Card-Komponente `Karte` ist in der Enumeration `ZustandKarte` definiert. Die möglichen Zustände der Karte sind `Z1_Karte` und `Z2_Karte`. Das Ende der

Assoziation zwischen `Karte` und `ZustandKarte` ist mit dem Stereotyp ≪status≫ annotiert. Für die Terminalkomponenten `konkretesTerminal1` und `konkretesTerminal2` gibt es zwei Möglichkeiten die möglichen Zustände zu modellieren. Die Enumeration kann für die gemeinsame Oberklasse `Terminal` definiert werden. In dem Fall ist die Angabe einer weiteren Enumeration für eine konkrete Terminalklasse nicht möglich. Die zweite Möglichkeit erlaubt es, eine Enumeration separat für jede konkrete Terminalklasse zu definieren. In diesem Fall haben beide Terminals verschiedene Zustände (`ZustandT1` und `ZustandT2`).

4.2.1.4. Modellierung von kryptographischen Daten

Neben den nicht-kryptographischen Daten ist auch die Modellierung von Daten, die verschlüsselt, signiert und gehasht werden oder über die ein MAC-Wert gebildet wird, notwendig. Hierfür werden die in Abschnitt 4.1.3 erläuterten Stereotypen verwendet.

Eine Klasse, die Klartextdaten repräsentiert, kann auch mehr als einen der Stereotypen ≪PlainData≫, ≪SignData≫ oder ≪HashData≫ tragen. Eine Objekt einer Klasse, die z.B. mit den Stereotypen ≪PlainData≫ sowie ≪HashData≫ annotiert ist, wird sowohl verschlüsselt als auch gehasht gespeichert. Jedes Assoziationsende darf allerdings nur maximal einen der Stereotypen ≪encrypted≫, ≪encryptedAsymm≫, ≪MAC≫, ≪signed≫ oder ≪hashed≫ tragen.

Abbildung 4.7 zeigt die Verwendung von kryptographischen Daten.

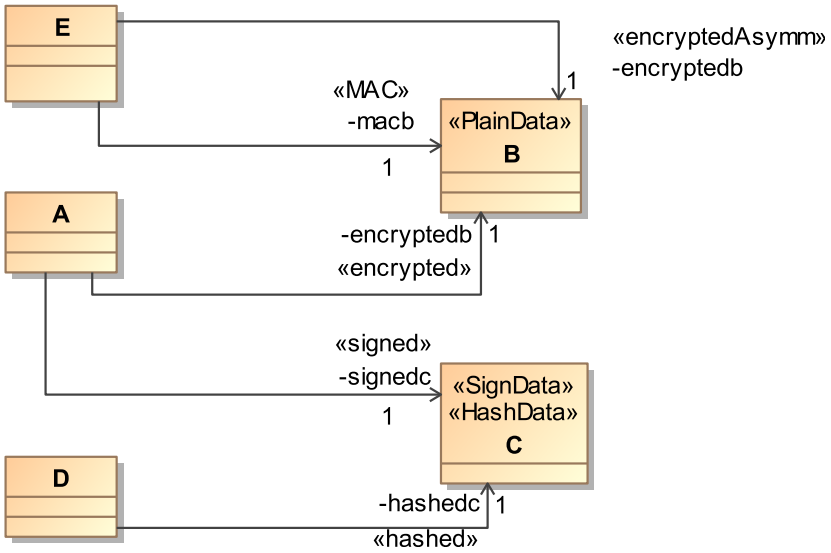

Abbildung 4.7.: Modellierung von kryptographischen Daten

Die Klasse `B` ist mit Stereotyp ≪PlainData≫ annotiert. Dieser Stereotyp gibt an, dass ein Objekt dieser Klasse (asymmetrisch oder symmetrisch) verschlüsselt oder zur Bildung eines MAC-Wertes verwendet werden kann. Die Klasse `E` speichert einen MAC-Wert über einem Objekt der Klasse `B` (das in `macb` gespeichert ist). Dies ist durch den Stereotyp ≪MAC≫ modelliert. Außerdem speichert die Klasse `E` ein verschlüsseltes Dokument (in `encryptedb`), das durch Verschlüsseln eines Objekts vom Typ `B` mit (einem `PublicKey`) erzeugt wurde.

Dies ist durch die Angabe des Stereotyps ≪encryptedAsymm≫, das das Assoziationsende `encryptedb` annotiert, modelliert. ≪encryptedAsymm≫ gibt an, dass ein asymmetrisches Verschlüsselungsverfahren verwendet wird. Der für die Verschlüsselung verwendete Schlüssel ist somit vom Typ `PublicKey`. Mit welchem konkreten Schlüssel das Objekt erzeugt wird, ist anhand des verschlüsselten Objekts nicht ablesbar. Wird das verschlüsselte Objekt während eines Protokolllaufs erzeugt, ist dies in den Aktivitätsdiagrammen modelliert. Wird das verschlüsselte Dokument jedoch während der Personalisierungsphase erzeugt und schon vor Inbetriebnahme der Anwendung auf einer Komponente gespeichert, ist anhand der UML-Modelle nicht erkennbar, mit welchem Schlüssel das verschlüsselte Dokument erzeugt wurde.

Die Klasse `A` speichert (in `encryptedb`) ebenfalls ein verschlüsseltes Objekt, das durch Verschlüsseln eines Objekts vom Typ `B` erzeugt wurde. Im Gegensatz zu der Klasse `E` ist hier das Objekt vom Typ `B` jedoch mit einem symmetrischen Schlüssel verschlüsselt worden. Dies ergibt sich aus der Verwendung des Stereotyps ≪encrypted≫, der für die Modellierung symmetrischer Verschlüsselung verwendet wird.

Die Klasse `C` ist mit den Stereotypen ≪SignData≫ und ≪HashData≫ annotiert. Dies gibt an, dass Objekte dieser Klasse sowohl signiert werden als auch der Hashwert über ihnen gebildet wird. Die Klasse `A` speichert (in `signedc`) eine Signatur des Objekts `C`. Dies ist durch den Stereotyp ≪signed≫ angegeben. Wie auch bei der Verschlüsselung ist im Klassendiagramm nicht angegeben, mit welchem `PrivateKey` die Signatur erzeugt wurde. Die Klasse `D` speichert (in `hashedc`) einen Hashwert, der über einem Objekt der Klasse `C` gebildet wurde. Dies ist durch Angabe des Stereotyps ≪hashed≫ am Assoziationsende `hashedc` modelliert.

Es stellt sich die Frage, warum die vordefinierten Sicherheitsdatentypen keine Datentypen für verschlüsselte Daten, MAC-Werte, digitale Signaturen und Hashwerte enthalten.So könnte man zum Beispiel den Datentyp `HashedData` definieren, der einen Hashwert repräsentiert. Anstelle der Verwendung der Stereotypen ≪HashData≫ und ≪hashed≫ könnten die Attribute, die einen Hashwert speichern, den Typ `HashedData` besitzen. Dies hat jedoch den Nachteil, dass nur die Tatsache, dass ein Hashwert gespeichert ist, modelliert wird. Aus welchen Datenklassen der Hashwert gebildet wurde, ist nicht ersichtlich. Dies ist jedoch für den Entwurf sowie die Sicherheit der Protokolle sehr wichtig und wird deshalb in SecureMDD explizit modelliert.

4.2.1.5. Modellierung von Zertifikaten

Zertifikate werden als eigene Klasse im Klassendiagramm definiert und müssen mit Stereotyp ≪Certificate≫ annotiert sein. Zu einer Klasse mit Stereotyp ≪Certificate≫ gehört immer genau eine Datenklasse, die ebenfalls im Klassendiagramm modelliert sein muss und die die im Zertifikat enthaltenen Daten enthält. Dies sind in der Regel ein öffentlicher Schlüssel sowie der Name des Zertifikatbesitzers. Die Angabe weiterer Informationen ist möglich. Diese Daten werden als Attribute in der Datenklasse definiert. Assoziationen zu weiteren Datenklassen sind nicht erlaubt. Die Datenklasse muss den Stereotyp ≪SignData≫ besitzen, da das Zertifikat die Signatur über den Daten enthält. Eine Klasse mit Stereotyp ≪Certificate≫ muss genau zwei gerichtete Assoziationen zu der Datenklasse besitzen, das gerichtete Ende muss jeweils auf die Datenklasse zeigen. Eine Assoziation speichert die Daten im Klartext, die andere Assoziation speichert die Signatur über die Daten und ist mit dem Stereotyp ≪signed≫ annotiert. Die Signatur wird vom Zertifikatherausgeber beim Erstellen des Zertifikats aus-

gestellt und kann mithilfe des öffentlichen Schlüssels des Herausgebers jederzeit verifiziert werden. Auf diese Weise ist es möglich, sich von der Echtheit des Zertifikats zu überzeugen.

Abbildung 4.8 zeigt die Definition eines Zertifikats mit Namen `Cert`.

Abbildung 4.8.: Modellierung von Zertifikaten

Die Daten des Zertifikats sind in der Klasse `CertData` gespeichert. Dies sind in diesem Fall ein öffentlicher Schlüssel `pukey`, ein Name `name` sowie eine `id`. Die Klasse darf beliebige viele Attribute, jedoch keine Assoziationen zu anderen Klassen haben. Die Klasse `Cert` speichert sowohl die Daten (in der Assoziation `data`) als auch die Signatur über diese Daten (in der Assoziation `sig` mit Stereotyp ≪signed≫).

4.2.1.6. Modellierung der verwendeten Nachrichtentypen

Die Kommunikation zwischen einer Smart Card und einem Terminal ist durch das Versenden von Nachrichten modelliert. Die verwendeten Nachrichtentypen sind als eigene Klassen im Klassendiagramm definiert. Alle Nachrichtenklassen müssen von einer abstrakten Klasse mit Stereotyp ≪Message≫ abgeleitet sein. Nachrichtenklassen können, wie die Komponenten- und Datenklassen, Attribute und Assoziationen zu weiteren Datenklassen besitzen. Die Regeln hierfür sind mit denen der Datenklassen identisch. Jede Komponentenklasse muss eine Usage-Dependency zu den Nachrichtenklassen haben. Dies gibt an, dass die Komponente die Nachrichten verschickt oder empfängt. Um nicht von jeder Komponente zu jeder Nachrichtenklasse eine Usage-Dependency modellieren zu müssen, reicht es aus, wenn jede Komponente eine Usage-Dependency zu der abstrakten Nachrichtenklasse mit Stereotyp ≪Message≫ besitzt. Auch wenn eine Komponentenklasse nur einige der Subklassen der Klasse mit Stereotyp ≪Message≫ verwendet, ist es möglich, eine Usage-Dependency zu der abstrakten Messageklasse zu modellieren. Die Berechnung, welche Nachrichtenklassen eine Komponente verwendet, basiert auf den Aktivitätsdiagrammen.

Abbildung 4.9 zeigt beispielhaft die Modellierung von Nachrichtenklassen.

Die Nachrichtenklassen `Nachricht1` und `Nachricht2` sind von der abstrakten Nachrichtenklasse `Message` abgeleitet, die mit Stereotyp ≪Message≫ annotiert ist. Die Komponentenklasse `Karte` verwendet diese Nachrichten, was durch die Usage-Dependency zwischen der Karte und der abstrakten Nachrichtenklasse modelliert ist. Die Nachrichtenklassen können zusätzlich Attribute und Assoziationen besitzen.

Die Modellierung der Nachrichtentypen als Klassen im Klassendiagramm bietet sich für sicherheitskritische Anwendungen an, da auf diese Weise für jeden Nachrichtentyp konkret angegeben werden kann, welche Eigenschaften (d.h. Attribute und Assoziationsenden) er besitzt

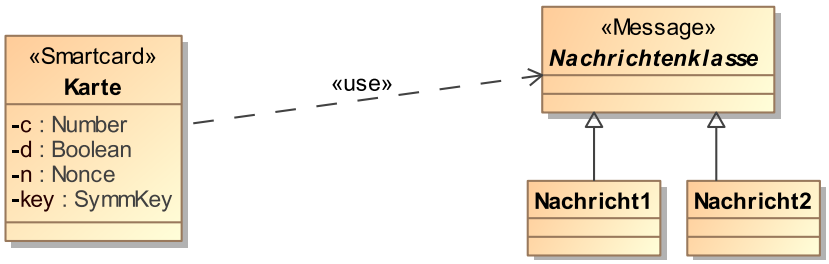

Abbildung 4.9.: Modellierung von Nachrichtenklassen

und von welchem Typ diese sind. Für kryptographische Daten, wie zum Beispiel Hashwerte, ist nicht nur ersichtlich, dass in einer Nachricht ein Hashwert enthalten ist, sondern auch über welche Daten(-typen) der Hashwert gebildet wurde. Dies ist für das Verständnis der Protokolle erforderlich und erleichtert den Entwurf.

Neben den Karten- und den Terminalkomponenten gibt es noch die Benutzer der Anwendung, die mit den Terminalkomponenten kommunizieren. Wie bei der Kommunikation zwischen einem Terminal und einer Karte, ist auch die Kommunikation zwischen einem Benutzer und einem Terminal nachrichtenbasiert. Die Modellierung ist gleich der Modellierung der Nachrichtenklassen. Jedoch müssen alle Benutzernachrichtenklassen von einer abstrakten Klasse mit Stereotyp ≪Usermessage≫ abgeleitet sein. Die Namen der Nachrichtentypen sind frei wählbar, es bietet sich jedoch an Benutzernachrichtenklassen mit einem „U“ beginnen zu lassen, um Usernachrichten von Nachrichten zwischen Komponenten unterscheiden zu können.

4.2.1.7. Angabe von Konstruktoren

Während eines Protokolllaufs werden Instanzen der Klassen, die im Klassendiagramm modelliert sind, gebildet. Dies gilt sowohl für die Datenklassen als auch die Nachrichtenklassen. Beispielsweise erzeugt eine Smart Card-Komponente während eines Protokolllaufs ein Klartextobjekt, verschlüsselt dieses und verschickt es anschließend in einem vorher erzeugten Nachrichtenobjekt. Die kryptographischen Protokolle sind im SecureMDD-Ansatz durch Aktivitätsdiagramme und die DSL MEL modelliert. Um in MEL Instanzen der modellierten Klassen erzeugen zu können, werden Konstruktoren für diese Klassen benötigt.

Abbildung 4.10 zeigt die Modellierung einer Klasse mit Konstruktordefinition.

Besitzt eine Klasse maximal eine Assoziation (bei der das gerichtete Assoziationsende von der Klasse weg zeigt), ergibt sich der Konstruktor der Klasse aus dem Namen der Klasse sowie den Argumenten mit denen das Objekt erzeugt wird. Die Reihenfolge der Argumente ergibt sich aus der Reihenfolge in der die Attribute in der Klasse definiert sind, gefolgt von dem Assoziationsende (falls vorhanden). Da der Konstruktoraufruf eindeutig ist, muss er nicht modelliert, sondern kann automatisch errechnet werden. Die Klasse `G` hat nur eine Assoziation zu einer anderen Klasse, bei der das gerichtete Assoziationsende von der Klasse `G` weg zeigt. Der Konstruktor der Klasse ist somit eindeutig berechenbar und muss im Modell nicht angegeben werden. Er lautet `G(a,b,j)`. Bei Aufruf des Konstruktors (in den Aktivitätsdiagrammen) muss für jedes Attribut und Assoziationsende einer Klasse ein Wert angegeben

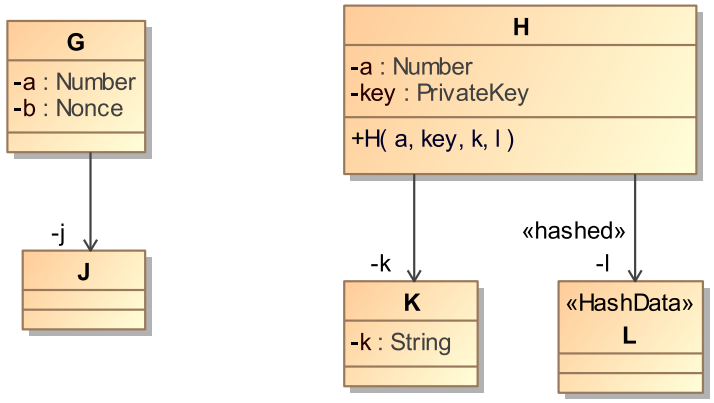

Abbildung 4.10.: Modellierung eines Konstruktors

werden, d.h. bei Erzeugen einer Instanz müssen immer alle Attribute und Assoziationsenden initialisiert werden.

Hat eine Klasse mehr als eine Assoziation (bei der das gerichtete Ende der Assoziation von der Klasse weg zeigt), ist die Reihenfolge der Argumente nicht mehr eindeutig. In diesem Fall muss im Klassendiagramm für diese Klasse ein Konstruktor definiert werden. Hierfür wird in der Klasse eine Operation mit dem Namen der Klasse definiert. Die Parameter der Operation sind die Attribute und Assoziationsenden der Klasse, d.h. die Methode hat so viele Parameter wie die Klasse Attribute und Assoziationsenden hat. Dies gibt vor, in welcher Reihenfolge die Attribute und Assoziationsenden beim Aufruf des Konstruktors angegeben werden müssen. Für jeden Parameter ist ein Name anzugeben, dieser muss dem Namen des Attributs oder Assoziationsendes entsprechen. Jede Klasse darf maximal eine Konstruktoroperation besitzen. Die Klasse `H` hat zwei Assoziationen, bei denen das gerichtete Assoziationsende von ihr weg zeigt. Somit muss für die Klasse ein Konstruktor angegeben werden. Die entsprechende UML-Operation hat den Namen `H` und Parameter für jedes Attribut und Assoziationsende. Die ersten beiden Parameter dienen der Initialisierung der Attribute und haben die Namen der Attribute (`a,key`). Der dritte Parameter ist für die Initialisierung des Assoziationsendes `k` zuständig. Der Name des Parameters ist `k`. Der vierte Parameter des Konstruktors initialisiert das Assoziationsende `l`. Der Konstruktor lautet somit `H(a,key,k,l)`. Die Parameter der Operation können in beliebiger Reihenfolge den gleichnamigen Attributen und Assoziationsenden zugewiesen werden.

4.2.1.8. Modellierung von manuell zu implementierenden Methoden

Die Modellierung mit SecureMDD ist auf Anwendungen, die auf kryptographischen Protokollen basieren, optimiert. Die Sprache MEL zur Erweiterung der Aktivitätsdiagramme stellt eine Reihe von vordefinierten Operationen zur Verfügung, die die Modellierung von kryptographischen Protokollen auf einfache Weise ermöglicht (siehe Kapitel 5). Es gibt jedoch Anwendungen, die zusätzlich anwendungsabhängige Operationen, zum Beispiel für komplexe Berechnungen, enthalten. Dies ist im SecureMDD-Ansatz durch die Verwendung von manuell zu implementierenden Operationen berücksichtigt. Wann immer möglich, sollte jedoch auf die

Verwendung von manuellen Methoden verzichtet und stattdessen die Modellierung der Operation mit Aktivitätsdiagrammen vorgezogen werden. Für den Fall, dass ein Methodenrumpf von Hand implementiert anstatt generiert (d.h. mit Aktivitätsdiagrammen modelliert) wird, kann eventuell die Sicherheit der Anwendung nicht mehr garantiert werden. In diesem Fall muss eine Quellcodeverifikation die gewünschten Eigenschaften sicherstellen.

Manuell zu implementierende Methoden müssen in der entsprechenden Klasse als Operationen deklariert werden. Die Operation muss mit dem Stereotyp ≪Manual≫ annotiert sein. Manuelle Operationen dürfen in Daten- sowie Komponentenklassen definiert werden. In einer Status-, Nachrichten- Benutzernachrichten- oder Konstantenklasse ist die Definition von manuellen Methoden nicht erlaubt. Eine manuelle Operation muss einen Namen sowie Parameter haben. Die Parameter müssen einen Namen und einen Typ besitzen und es müssen sowohl die Eingabe- als auch der Ausgabeparameter angegeben werden. Hat eine Methode keinen Rückgabewert, muss dieser nicht angegeben werden. Manuelle Methoden können statisch sein, müssen sie aber nicht. Statische Operationen sind in UML unterstrichen dargestellt. Aufrufe von manuellen Methoden können mit Aktivitätsdiagrammen modelliert werden. Dies ist detailliert in Abschnitt 4.4 beschrieben.

Abbildung 4.11 zeigt die Modellierung von manuellen Methoden.

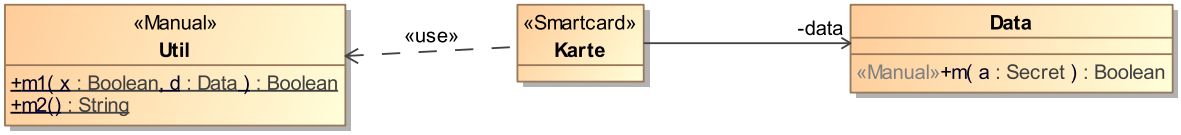

Abbildung 4.11.: Modellierung von manuellen Methoden

Die Klasse `Data` definiert die manuelle Methode `m` mit Eingabeparameter `a` (vom Typ `Secret`) und Rückgabetyp `Boolean`. Diese Methode kann auf einer Instanz der Klasse `Data` aufgerufen werden.

Wann immer möglich, sollten manuelle Methoden in der Klasse, in der sie benötigt werden, definiert sein. In seltenen Fällen gibt es jedoch Methoden, die von mehreren Klassen benötigt werden bzw. deren Funktionalität keiner Klasse zugeordnet werden kann. Diese Methoden können in einer separaten Klasse deklariert werden. Wenn möglich, sollte aus Designgründen hierauf jedoch verzichtet werden.

Eine Klasse, deren einziger Zweck darin besteht manuelle Methoden zu deklarieren, muss mit Stereotyp ≪Manual≫ annotiert sein. Da die Klasse eine Hilfsklasse ist, bietet es sich an ihr den Namen `Util` zu geben. Prinzipiell darf jedoch ein beliebiger Name vergeben werden. Die manuellen Methoden müssen ebenfalls als Operationen definiert sein, einen Namen haben sowie (falls vorhanden) Ein- und Ausgabeparameter definieren. Zusätzlich müssen die Methoden einer mit ≪Manual≫ annotierten Klasse statisch sein. Die Methoden selber müssen nicht mit Stereotyp ≪Manual≫ gekennzeichnet sein. Eine Klasse, die Methoden der mit ≪Manual≫ annotierten Klasse aufruft, muss eine Usage-Dependency zu dieser Klasse haben. Dies gibt an, dass die annotierte Klasse der anderen Klasse bekannt ist.

Die Klasse `Util` definiert zwei manuelle Methoden. Sie ist deshalb mit Stereotyp ≪Manual≫ annotiert. Beide Methoden müssen statisch sein. Die Methode `m1` hat zwei Eingabeparameter, vom Typ `Boolean` sowie vom Typ `Data`, und gibt einen Wert vom Typ `Boolean` zurück. Die

Methode `m2` hat keine Eingabeparameter und gibt einen Wert vom Typ `String` zurück. Die Smart Card-Komponentenklasse `Karte` ruft Methoden dieser Klasse auf, d.h. die Methoden müssen der Komponentenklasse bekannt sein. Dies ist durch die Usage-Dependency zu der Klasse `Util` modelliert.

4.2.1.9. Initialisierung von Daten

Vor Produktivsetzung einer Anwendung müssen die Daten der Komponenten initialisiert werden. Für Anwendungen, an denen Smart Cards beteiligt sind, ist vor Auslieferung der Karten an den Kunden eine Personalisierungsphase vorgesehen. In dieser werden zum Beispiel kryptographische Schlüssel, aber auch persönliche Daten (wie zum Beispiel Kreditkartendaten oder die PIN) auf die Karte gespielt und dort persistent gespeichert. Weiterhin muss der Zustand der Karte auf einen Anfangszustand, meist der Zustand `IDLE`, gesetzt werden. Ebenso müssen die Terminalkomponenten initialisiert werden. Auch auf ihnen müssen beispielsweise kryptographische Schlüssel oder weitere Daten gespeichert werden, die bei der Produktivsetzung der Anwendung bereits bekannt sein müssen.

SecureMDD bietet eine Möglichkeit anzugeben, welche Daten vor der Produktivsetzung initialisiert werden können bzw. müssen. Der Zustand einer Komponente muss immer initialisiert werden, alle weiteren Attribute und Assoziationsenden können initialisiert werden. Attribute und Assoziationsenden die initialisiert werden sollen, müssen mit Stereotyp ≪Initialize≫ annotiert sein.

Das Annotieren mit dem Stereotyp ≪Initialize≫ bewirkt, dass während der Codegenerierung ein Mechanismus generiert wird, der die Initialisierung der annotierten Attribute und Assoziationsenden vor Inbetriebnahme der Anwendung ermöglicht. Die Initialisierung nichtannotierter Attribute und Assoziationsenden ist somit nicht möglich.

Attribute und Assoziationsenden, die nicht mit Stereotyp ≪Initialize≫ annotiert sind, werden bei der Codegenerierung bzw. der Generierung des formalen Modells mit Defaultwerten vorinitialisert.

4.2.1.10. Definition von Konstanten

Da die Nachrichten kryptographischer Protokolle oftmals Konstanten enthalten, sieht die Modellierung die Definition von Konstanten in einer Klasse vor, die mit dem Stereotyp ≪Constant≫ annotiert ist. Die eigentlichen Konstanten sind als Attribute dieser Klasse modelliert. Konstanten können vom Typ `Number`, `String` oder `Boolean` sein. Ist für eine Konstante kein Typ angegeben, hat diese den Defaulttyp `Number`. Des Weiteren ist es möglich, einer Konstante einen Wert zuzuweisen (durch Angabe eines Defaultwerts im Modellierungstool). Da für die meisten Anwendungen der konkrete Wert der Konstante nicht von Bedeutung ist, kann dieser für Konstanten vom Typ `Number` und `Boolean` weggelassen werden. Konstanten vom Typ `Number` werden dann aufsteigend, beginnend mit null nummeriert. Konstanten vom Typ `Boolean` bekommen den Wert `false`. Für Konstanten vom Typ `String` muss ein Wert angegeben werden.

4.2.2. Deploymentdiagramme

Die Deploymentdiagramme beschreiben die Kommunikationsstruktur der Anwendung sowie die Möglichkeiten, die ein Angreifer hat, um Nachrichten abzuhören, zu manipulieren oder zu unterdrücken. In Abschnitt 4.2.2.1 ist die Modellierung der Kommunikationsstruktur erläutert, Abschnitt 4.2.2.2 beschreibt die Modellierung der Angreiferfähigkeiten.

4.2.2.1. Kommunikationsstruktur

In der Regel ist nicht jede Komponente in der Lage, mit jeder beliebigen anderen Komponente zu kommunizieren. Die Benutzer kommunizieren nur mit (einigen) Terminalkomponenten. Sie geben über eine Benutzeroberfläche Anweisungen an das Terminal und erhalten eine Antwort des Systems. Beispielsweise gibt ein Benutzer an das Kopiergerät die Anweisung Kopien anzufertigen. Nach Beenden des Kopiervorgangs bekommt der Benutzer vom Kopiergerät die Rückmeldung, dass der Vorgang erfolgreich war bzw. erhält er im Fehlerfall eine Fehlermeldung. Die Kartenkomponenten können ebenfalls nur mit Terminalkomponenten kommunizieren, da nur diese ein Kartenlesegerät besitzen. Die Kommunikationskanäle der Anwendung sowie die Komponenten, zwischen denen diese Kanäle bestehen, sind im Deploymentdiagramm definiert.

Die Komponenten des zu entwickelnden Systems sind als Knoten (`Node`) dargestellt. Diese tragen den Namen der Komponente aus dem Klassendiagramm sowie den entsprechenden Stereotyp. Da die Metaklasse `Node` eine Subklasse der Metaklasse `Class` ist, sind die Stereotypen, die die Metaklasse `Class` erweitern auch auf `Nodes` anwendbar. Ein Kommunikationskanal ist in UML durch einen `CommunicationPath` modelliert. Dieser verbindet zwei Knoten und gibt an, dass diese miteinander Nachrichten austauschen. Die Kommunikationskanäle müssen gerichtet sein und geben an, welche Komponente die Kommunikation initiiert. Ein gerichteter Kommunikationskanal, der z.B. von der Komponente `Benutzer` auf die Komponente `Terminal` zeigt, gibt an, dass der Benutzer die Kommunikation mit dem Terminal startet.

Abbildung 4.12 zeigt in einem Deploymentdiagramm die Komponenten `Benutzer`, `Terminal` und `Karte`, die als Knoten modelliert sind sowie die Kommunikationskanäle.

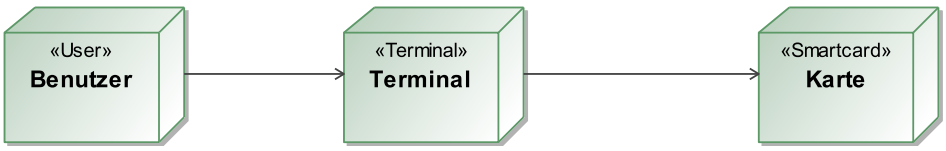

Abbildung 4.12.: Komponenten und Kommunikationsstruktur im Deploymentdiagramm

Es gibt zwei Regeln für die Initiierung der Kommunikation:

- Die Kommunikation zwischen einem Benutzer und einem Terminal wird immer von dem Benutzer begonnen. Das bedeutet, dass das gerichtete Ende des Kommunikationskanals auf das Terminal zeigen muss. Dies liegt daran, dass der Benutzer dem Terminal sagt, welche Funktion das Terminal ausführen soll, zum Beispiel Aufladen der Kopierkarte.

- Die Kommunikation zwischen einem Terminal und einer Smart Card wird immer von dem Terminal initiiert. Dies liegt daran, dass eine Smart Card aus technischen Gründen keine Nachrichten verschicken, sondern nur auf empfangene Nachrichten antworten kann. Das bedeutet, dass bei Kommunikationskanälen zwischen Terminals und Smart Cards das gerichtete Ende auf die Karte zeigen muss.

Ist für einen Komponententyp im Klassendiagramm eine abstrakte Oberklasse definiert und sind von dieser konkrete Komponentenklassen abgeleitet, gibt es zwei Möglichkeiten das Deploymentdiagramm anzugeben. Die eine Möglichkeit ist, nur die Oberklasse im Deploymentdiagramm zu modellieren. Die für die Oberklasse definierten Kommunikationskanäle gelten dann für jede Subklasse. Alternativ können die Subklassen sowie ihre Kommunikationskanäle explizit modelliert werden. In diesem Fall muss die Kommunikationsstruktur für alle Subklassen angegeben werden. Unterscheiden sich die Kommunikationsstrukturen der Subklassen einer Anwendung voneinander, müssen im Deploymentdiagramm Knoten für die konkreten Komponenten angegeben werden. Die Angabe der abstrakten Oberklasse ist dann nicht möglich.

Es gibt außerdem die Möglichkeit, dass eine Anwendung aus mehreren Smart Card-Komponenten besteht, die mit dem gleichen Terminal kommunizieren. Ein Beispiel hierfür ist die elektronische Gesundheitskarte, bei der die Terminalkomponente Arzt-PC sowohl mit der Gesundheitskarte als auch dem Heilberufsausweis des Arztes kommuniziert. In diesem Fall hat das Deploymentdiagramm je einen Knoten für die Gesundheitskarte sowie für den Heilberufsausweis und je einen gerichteten Kommunikationskanal zwischen dem Terminal und jedem der beiden Smart Card-Knoten. Technisch ist dies gleichbedeutend damit, dass das Terminal mehrere Kartenleser, nämlich einen für jede Verbindung zu einer Smart Card, besitzt.

In Abbildung 4.13 (links) gibt es die Terminalkomponente `Terminal2`, die mit den beiden Kartenkomponenten `Karte1` und `Karte2` kommuniziert. Da das Terminal zwei Kommunikationskanäle zu Smart Card-Komponenten besitzt, sind für die Terminalkomponente zwei Kartenleser notwendig.

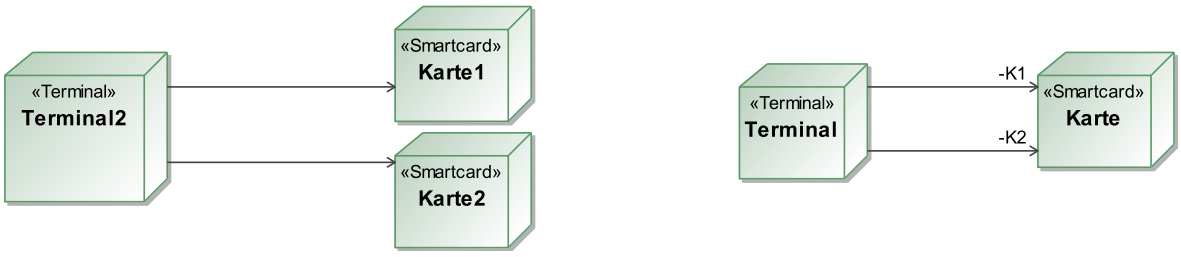

Abbildung 4.13.: Modellierung eines Terminals und mehreren Kartenkomponenten im Deploymentdiagramm

Ein weiterer Spezialfall ist der, dass das Terminal zwar mit zwei Karten kommuniziert, diese jedoch eine Instanz der gleichen Smart Card-Komponente sind. Ein Beispiel hierfür ist das Bezahlsystem Mondex, bei dem ein Geldbetrag von einer Smart Card auf eine andere gebucht wird. Jede Karte kann dabei als diejenige, auf die ein Betrag aufgebucht wird als auch als eine, von der ein Betrag abgebucht wird, fungieren. Beide Karten sind also vom selben Typ und werden, je nach Protokolllauf zum Abbuchen oder zum Aufbuchen verwendet. Die

Modellierung hierfür ist in Abbildung 4.13 (rechts) dargestellt. Die Komponente `Terminal` kommuniziert mit zwei Instanzen der Smart Card-Komponente `Karte`. Dies ist modelliert durch zwei gerichtete Kommunikationspfade zwischen dem Terminal und der Karte. Um die Pfade auseinanderhalten zu können, muss das gerichtete Ende mit einem eindeutigen Namen (`K1` und `K2`) versehen sein. Dieses Ende wird in SecureMDD `Port` genannt. Da das Terminal zwei Kommunikationskanäle zu einer Smart Card-Komponente besitzt, sind für die Terminalkomponente zwei Kartenleser notwendig.

Im Deploymentdiagramm ist nur die Kommunikation der Komponenten modelliert wie sie bei korrektem Gebrauch der Anwendung vorgesehen ist. Die Modellierung der Kommunikation zwischen Komponenten, die durch fehlerhaften Gebrauch der Komponenten entsteht, ist nicht modelliert. Hat eine Anwendung mehrere Terminal- und Smart Card-Komponenten ist es natürlich technisch möglich, dass eine Smart Card in einen Kartenleser an einem Terminal gesteckt wird, das für diese Karte nicht vorgesehen ist. Beispielsweise enthält die Anwendung der elektronischen Gesundheitskarte den Arzt-PC, der einen Kartenleser für die Gesundheitskarte sowie einen für den Heilsberufsausweis des Arztes vorsieht. Außerdem gibt es den Heilberufsausweis der Apotheker, der sich bei korrektem Gebrauch der Anwendung nicht in einem Arzt-PC, sondern nur in einem Apotheken-PC befindet. Aus diesem Grund enthält das Deploymentdiagramm keinen Kommunikationspfad zwischen dem Heilberufsausweis des Apothekers und dem Arzt-PC. Trotzdem muss beim Entwurf der Anwendung natürlich darauf geachtet werden, dass die Anwendung auch sicher ist, wenn der Heilberufsausweis eines Apothekers in den Kartenleser eines Arzt-PCs gesteckt wird.

4.2.2.2. Modellierung der Fähigkeiten des Angreifers

Zusätzlich sind im Deploymentdiagramm die Fähigkeiten des Angreifers beschrieben. Ein Angreifer hat die Möglichkeit, Nachrichten, die über einen Kommunikationskanal gesendet werden, zu lesen, zu manipulieren oder die Nachricht zu unterdrücken. Für jeden Kommunikationskanal kann deshalb angegeben werden, ob der Angreifer die über den Kanal gesendeten Nachrichten abhören, manipulieren oder unterdrücken kann.

Durch Annotation des Stereotyps ≪Threat≫ kann für jeden Kanal angegeben werden, ob der Angreifer diesen manipulieren kann oder nicht. Die Manipulation einer Nachricht besteht darin, eine Nachricht zu lesen, sie zu ändern und anschließend zu senden. Dies lässt sich durch die Tags `read` und `send` ausdrücken. Die Angabe der Fähigkeiten des Angreifers wird zu Dokumentationszwecken sowie für die Generierung der formalen Spezifikation verwendet.

Abbildung 4.14 zeigt beispielhaft die Modellierung von Angreiferfähigkeiten.

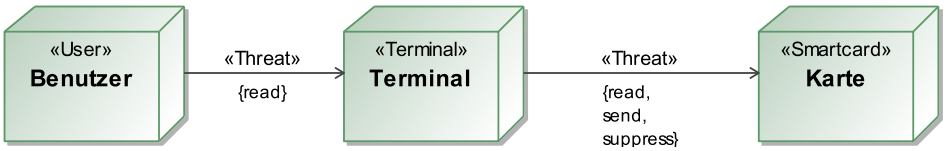

Abbildung 4.14.: Modellierung der Angreiferfähigkeiten

In dem Beispiel kann die Kommunikation zwischen dem Benutzer und dem Terminal von einem Angreifer abgehört werden. Dies ist durch Annotation des Stereotyps ≪Threat≫ und

Angabe des Tags `read` modelliert. Die Kommunikation zwischen dem Terminal und der Karte kann ein Angreifer abhören, Nachrichten über den Kanal senden sowie Nachrichten unterdrücken. Modelliert ist dies durch Annotation des Stereotyps ≪Threat≫ mit den Tags `read`, `send` und `suppress`.

Die Kommunikationskanäle zwischen einem Benutzer und einem Terminal entsprechen den Kanälen, über die ein Benutzer eine Eingabe an das Terminal macht. Als ≪Threat≫ ist an dieser Stelle maximal das Lesen von Nachrichten sinnvoll (Tag `read`). Dies entspricht einem Angreifer, der dem Benutzer über die Schulter schaut und so dessen Eingaben mitliest. Das Senden eigener Nachrichten (`send`) oder das Unterdrücken von Nachrichten (`suppress`) ist an dieser Stelle nicht sinnvoll und wird nicht unterstützt.

Enthält das Deploymentdiagramm eine abstrakte Oberklasse als Knoten und sind die Kommunikationskanäle dieses Knotens mit Stereotyp ≪Threat≫ annotiert, gelten die Angreiferfähigkeiten automatisch für alle Kommunikationskanäle der konkreten Komponentenklassen, die sich durch Ersetzen der abstrakten Oberklasse durch die konkrete Klasse ergeben.

Bei der Betrachtung und Verifikation von kryptographischen Protokollen ist die Annahme eines Dolev-Yao Angreifers [51] üblich. Dieser hat vollen Zugriff (read, send und suppress) auf alle Kommunikationskanäle. Der SecureMDD-Ansatz ermöglicht an dieser Stelle durch konkrete Angaben für jeden Kommunikationskanal eine feingranularere Modellierung des Angreifers als das Dolev-Yao Modell.

4.3. Statische Modellierung der Kopierkartenanwendung

In diesem Abschnitt wird der statische Teil der plattformunabhängigen Modellierung am Beispiel der Kopierkartenanwendung vorgestellt. Abschnitt 4.3.1 erläutert die Klassendiagramme der Anwendung, in Abschnitt 4.3.2 wird das Deploymentdiagramm erläutert.

Einige Modellierungselemente kommen in der Modellierung der Kopierkartenanwendung nicht vor, sind aber in den UML-Modellen der elektronischen Gesundheitskarte (siehe Abschnitt 11.2) und anderer Fallstudien (siehe Abschnitt 3.3.3 bzw. auf der Projekt-Webseite[1]) enthalten. Speziell sind dies die Verwendung von symmetrischer Verschlüsselung (EGK und Mondex), asymmetrischer Verschlüsselung (EGK und Geldkarte), digitalen Signaturen (EGK, Geldkarte und Altersverifikation), MAC-Werten (elektronischer Reisepass), Zertifikaten (EGK, Geldkarte und Altersverifikation). Beispiele für die Angabe von Konstruktoren finden sich in den Modellen der EGK, des Reisepasses, der Geldkarte sowie der Altersverifikation. Listen, d.h. Assoziationen zu Datenklassen, deren gerichtetes Ende eine Multiplizität größer als eins hat, werden in den Modellen der EGK sowie Mondex verwendet. Manuell zu implementierende Methoden finden sich in dem Modell des elektronischen Reisepasses sowie der Altersverifikation.

4.3.1. Die Klassendiagramme der Kopierkartenanwendung

Abbildung 4.15 zeigt das Klassendiagramm der Kopierkartenanwendung. Die Anwendung besteht aus einer Komponente für die eigentliche Kopierkarte sowie aus zwei Terminalkom-

[1] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/`

ponenten. Die Kopierkarte ist durch eine Klasse namens `Copycard` modelliert, die mit dem Stereotyp ≪Smartcard≫ annotiert ist. Bei den Terminals muss man zwischen einem Terminaltyp zum Aufladen der Karte sowie einem Typ, der sich an ein Kopiergerät anschließen lässt und zum Bezahlen der Kopien verwendet wird, unterscheiden. Da beide Terminaltypen einige Attribute gemeinsam haben, gibt es eine abstrakte Oberklasse namens `Terminal`, die mit dem Stereotyp ≪Terminal≫ annotiert ist. Die Komponente zum Aufladen einer Karte ist durch eine Klasse namens `DepositMachine` definiert. Die Komponente zum Abbuchen des Betrags für gemachte Kopien von der Karte ist durch die Klasse `CopyingMachine` modelliert. Beide Klassen sind als Subklasse der abstrakten Klasse `Terminal` definiert.

In der aus dem plattformunabhängigen Modell erzeugten Implementierung sowie der formalen Spezifikation gibt es mehrere Instanzen dieser Klassen, d.h. es kann beliebig viele Kopierkarten sowie Terminals geben.

Außerdem gibt es eine Klasse `CardOwner`, die die Besitzer der Kopierkarten und Nutzer der Anwendung repräsentiert. Diese Klasse ist mit Stereotyp ≪User≫ annotiert.

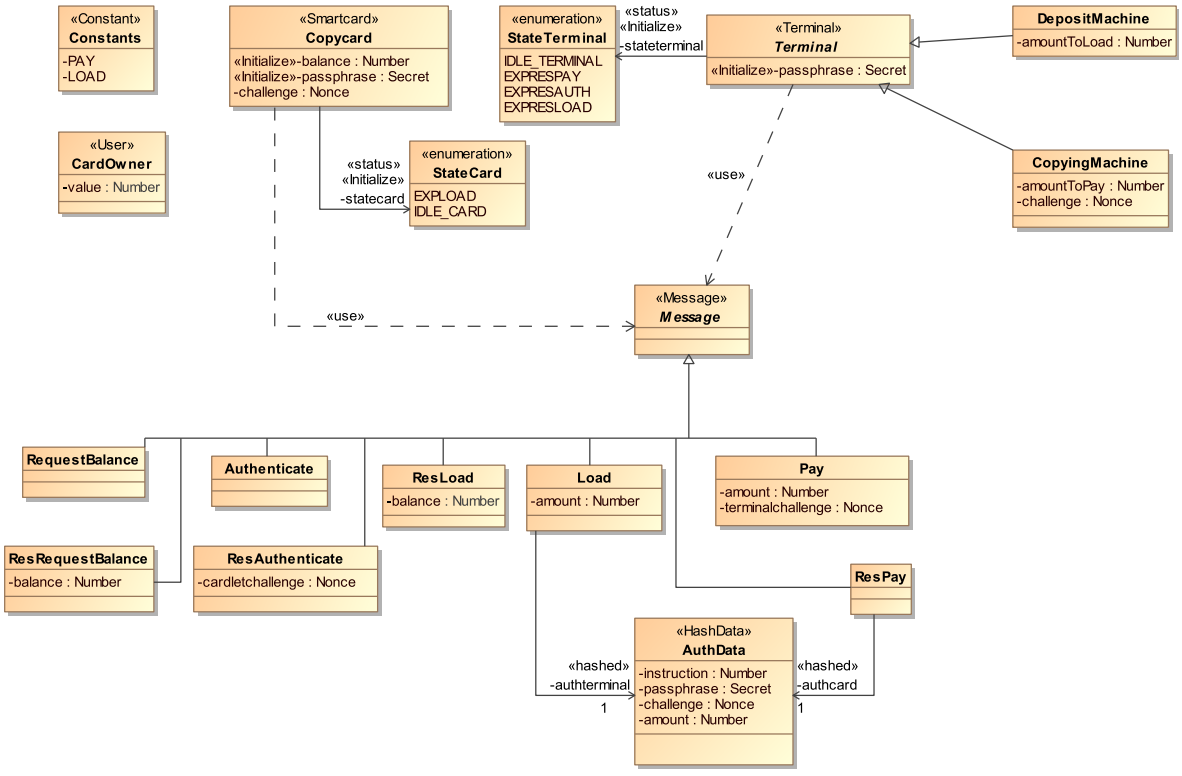

Abbildung 4.15.: Klassendiagramm für die Kopierkartenanwendung

Die möglichen Zustände der Smart Card-Klasse `Copycard` sind in der Enumerationklasse `StateCard` modelliert. Diese Enumeration listet alle Zustände auf, in der sich die Smart Card-Komponente während der Protokollläufe befinden kann. Dies sind die Zustände EXPLOAD (expecting a Load message) und IDLE_Card. Die Klasse `Copycard` hat eine gerichtete Assoziation auf die Klasse `StateCard`. Das gerichtete Assoziationsende ist mit den Stereotypen ≪status≫ (weil es sich um den Zustand handelt) und ≪Initialize≫ (weil

der Zustand in der Personalisierungsphase initialisiert werden muss) annotiert. Die möglichen Zustände der Terminalkomponenten `DepositMachine` und `CopyingMachine` sind in der Enumerationklasse `StateTerminal` zusammengefasst. Diese wird von der gemeinsamen Oberklasse `Terminal` assoziiert. Alternativ wäre es auch möglich, für beide konkrete Terminalklassen eine eigene Enumerationklasse für den Zustand zu definieren. Die möglichen Zustände der Terminals sind IDLE_TERMINAL, EXPRESPAY (expecting a ResPay message), EXPRESAUTH (expecting a ResAuth message) und EXPRESLOAD (expecting a ResLoad message). Das Assoziationsende der Assoziation zwischen der Klasse `Terminal` und der Enumeration `StateTerminal` ist ebenfalls mit den Stereotypen ≪status≫ und ≪Initialize≫ annotiert.

Die Klasse `Copycard` hat die Attribute `balance` vom Typ `Number` zum Speichern des Kontostands der Karte sowie ein Attribut `passphrase` vom Typ `Secret` für die geheime Passphrase, die nur den echten Kopierkarten (und den echten Terminals) bekannt sein darf. Des Weiteren besitzt die `Copycard` ein Attribut `challenge` vom Typ `Nonce`, das zur Speicherung einer Nonce dient. In unserem Fall wird die Nonce für eine *Challenge-Response Authentisierung* des Ladeterminals benötigt. Die Attribute `balance` und `passphrase` werden vor Auslieferung der Karten an die Kunden, während der Personalisierungsphase der Karte, initialisiert.

Die Klasse `CardOwner` hat ein Attribut mit Namen `value` vom Typ `Number`. Dies speichert den Geldbetrag, den ein Benutzer während eines Protokolllaufs auf die Karte laden bzw. für den er Kopien anfertigen möchte.

Die abstrakte Klasse `Terminal` besitzt ebenfalls ein Attribut `passphrase` für die geheime Passphrase. Da sowohl die Lade- als auch die Bezahlterminals dieses Geheimnis kennen müssen, ist es in der Klasse `Terminal` gespeichert. Die Klasse `DepositMachine` hat zusätzlich ein Attribut `amountToLoad` vom Typ `Number`, um den aktuell auf die Karte zu ladenden Betrag (für den Zeitraum während Geld auf die Karte geladen wird) zu speichern. Die Klasse `CopyingMachine` besitzt ein Attribut `challenge` zum Speichern der aktuell verwendeten Nonce beim Bezahlen mit der Karte, da sich die Karte hier mithilfe eines *Challenge-Response Verfahrens* gegenüber dem Terminal als echt nachweisen muss. Weiterhin speichert das Bezahlterminal den aktuell zu zahlenden Betrag während eines Protokolllaufs in einem Attribut namens `amountToPay`. Die Komponenten der Kopierkartenanwendung besitzen keine Assoziationen zu Datenklassen. Die einzige Datenklasse, die in dieser Fallstudie vorkommt, ist die Klasse `AuthData`, die zu den Nachrichtenklassen `Load` und `ResPay` assoziiert ist. Sie enthält Klartextdaten, über denen später, während der Protokollläufe, ein Hashwert gebildet wird. Deshalb ist die Klasse mit dem Stereotyp ≪HashData≫ annotiert.

An dieser Stelle wird deutlich, dass es nicht möglich ist, die einzelnen Diagramme unabhängig voneinander in sequentieller Reihenfolge zu entwickeln. Die Klassendiagramme beinhalten bereits Informationen, die man nur kennt, wenn man sich bereits über das Design der kryptographischen Protokolle Gedanken gemacht hat. Dies bedeutet, dass die Erstellung der Diagramme parallel erfolgen muss. So benötigt man beim Erstellen der Klassendiagramme bereits Kenntnis der Sequenzdiagramme und (in der Regel) auch der Aktivitätsdiagramme.

Die Nachrichtentypen für die Protokolle zum Abfragen des Kontostands (`RequestBalance` und `ResRequestBalance`), zum Aufladen der Kopierkarte (`Authenticate`, `ResAuthenticate`, `Load` und `ResLoad`) sowie zum Anfertigen von Kopien (`Pay` und `ResPay`) sind jeweils als eigene Klassen im Klassendiagramm modelliert. Die Nach-

richt `ResAuthenticate` beispielsweise enthält eine Nonce. Dies ist durch das Attribut `cardletchallenge` vom Typ `Nonce` modelliert. Der Nachrichtentyp `Load` beschreibt die Nachricht, die das Laden eines Geldbetrags auf eine Kopierkarte zur Folge hat. Diese Klasse hat ein Attribut `amount` vom Typ `Number` (das den aktuell auf die Karte zu ladenden Geldbetrag speichert) sowie eine gerichtete Assoziation zu der Datenklasse `AuthData`. Das Assoziationsende ist mit Stereotyp ≪hashed≫ annotiert, d.h. der Nachrichtentyp enthält einen Hashwert, der über einem Objekt der Klasse `AuthData` gebildet wird. Dieser Hashwert stellt sicher, dass nur von einem echten Terminal aus Geld auf die Karte geladen werden kann. Die Datenklasse `AuthData` kapselt die zu hashenden Klartextdaten, bestehend aus den vier Attributen `instruction` (eine Konstante, die den Typ der Nachricht (Load) nochmal explizit speichert), `passphrase` (das Geheimnis, das nur echte Karten und Terminals kennen), `challenge` (eine im vorherigen Schritt neu erzeugte Nonce um Replay Attacken zu verhindern) und `amount` (für den zu ladenden Betrag). Alle Nachrichtentypen sind von der abstrakten Klasse `Message` abgeleitet, die mit Stereotyp ≪Message≫ annotiert ist. Die abstrakte Klasse `Terminal` sowie die Klasse `Copycard` haben eine Usage-Dependency zu der abstrakten Klasse `Message`.

Die Benutzernachrichten sind in einem separaten Klassendiagramm definiert. Sie sind in Abbildung 4.16 dargestellt.

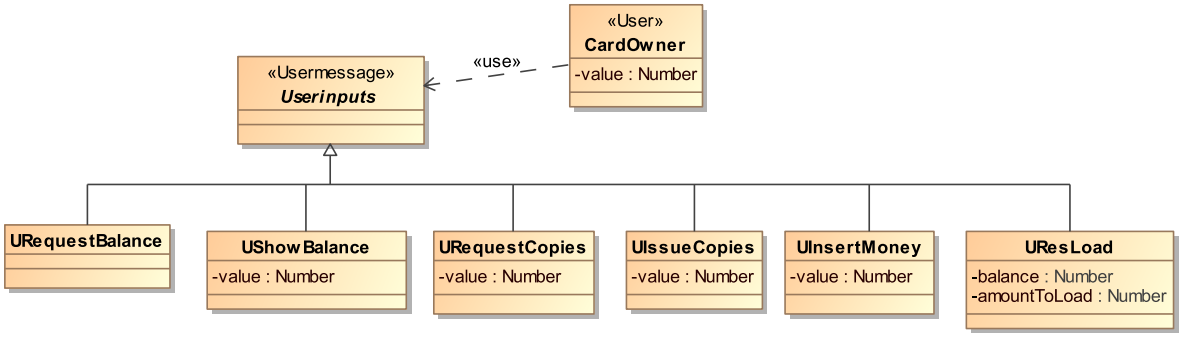

Abbildung 4.16.: Definition der Benutzernachrichten

Die Klasse `URequestBalance` bewirkt das Abfragen des Kontostands von der Karte, die Nachricht `UShowBalance` mit Attribut `amount` ist die entsprechende Rückantwort und enthält den aktuellen Kontostand. Die Nachricht `URequestCopies` modelliert die Anfrage des Benutzers Kopien anzufertigen. Das Attribut `value` gibt den hierfür von der Karte abzubuchenden Betrag an. Die Nachricht `UIssueCopies` signalisiert dem Benutzer, dass der Betrag `value` von der Karte abgebucht wurde und nun Kopien angefertigt werden können. Die Nachricht `UInsertMoney` wird verwendet, um dem Ladeterminal mitzuteilen, dass der Betrag `value` auf die Karte geladen werden soll. Diese Nachricht modelliert das Einwerfen des entsprechenden Betrags in das Ladeterminal. Alle Benutzernachrichtentypen sind ebenfalls von einer abstrakten Klasse abgeleitet, die mit dem Stereotyp ≪Usermessage≫ annotiert ist. Dies definiert, dass alle abgeleiteten Klassen Benutzernachrichten sind.

Weiterhin werden für die Protokolle zum Aufladen der Karte und zum Bezahlen von Kopien jeweils eine Konstante (`LOAD` und `PAY`) benötigt. Diese sind in der Klasse mit dem Namen `Constants` definiert. Die Klasse muss mit dem Stereotyp ≪Constant≫ annotiert sein. Die

Konstanten sind als Attribute ohne Angabe eines Typs definiert. Da die Typangabe fehlt, wird ihnen bei der Generierung des Codes sowie der formalen Spezifikation standardmäßig der Typ `Number` gegeben.

4.3.2. Das Deploymentdiagramm der Kopierkartenanwendung

Abbildung 4.17 zeigt zwei mögliche Deploymentdiagramme für die Kopierkartenanwendung. Beide stellen denselben Sachverhalt dar. Im linken Diagramm ist nur die Oberklasse `Terminal` vorhanden. Für jede abgeleitete Terminalklasse gilt somit die dort festgelegte Kommunikationsstruktur sowie die festgelegten Angreiferfähigkeiten. Im rechten Diagramm sind die konkreten Terminalklassen separat modelliert.

Die Komponenten sowie die Benutzer der Anwendung sind als Knoten im Deploymentdiagramm modelliert. Im linken Diagramm sind dies die Kartenbesitzer `CardOwner`, die abstrakte Klasse `Terminal` sowie die `Copycard`. Im rechten Diagramm ist die abstrakte Klasse `Terminal` jeweils durch ihre beiden Subklassen `DepositMachine` und `CopyingMachine` ersetzt. Die Kartenbesitzer kommunizieren mit den Ladeterminals und den Kopiergeräten, um z.B. anzugeben, welcher Betrag aufgeladen werden soll oder wie viele Kopien angefertigt werden sollen. Es gibt deshalb einen Kommunikationspfad zwischen dem Knoten `CardOwner` und dem Knoten `Terminal` (links) bzw. zwischen dem `CardOwner` und dem Knoten `DepositMachine` sowie dem Knoten `CopyingMachine` (rechts). Da die Kommunikation von den Kartenbesitzern initiiert wird, zeigen die gerichteten Kommunikationskanäle auf die Terminalknoten. Die Kommunikation zwischen den Terminals und den Kopierkarten ist ebenfalls durch Kommunikationskanäle modelliert. Im Diagramm links ist dies durch einen Kommunikationspfad zwischen dem `Terminal`- und dem `Copycard`-Knoten angegeben. Im Diagramm rechts ist dies durch Kommunikationskanäle von dem Knoten des Ladeterminals sowie des Kopiergeräts zu der `Copycard` modelliert. Da die Kommunikation zwischen einem Terminal und einer Smart Card immer von einem Terminal initiiert wird, zeigen die gerichteten Enden der Kommunikationspfade jeweils auf den Knoten der `Copycard`.

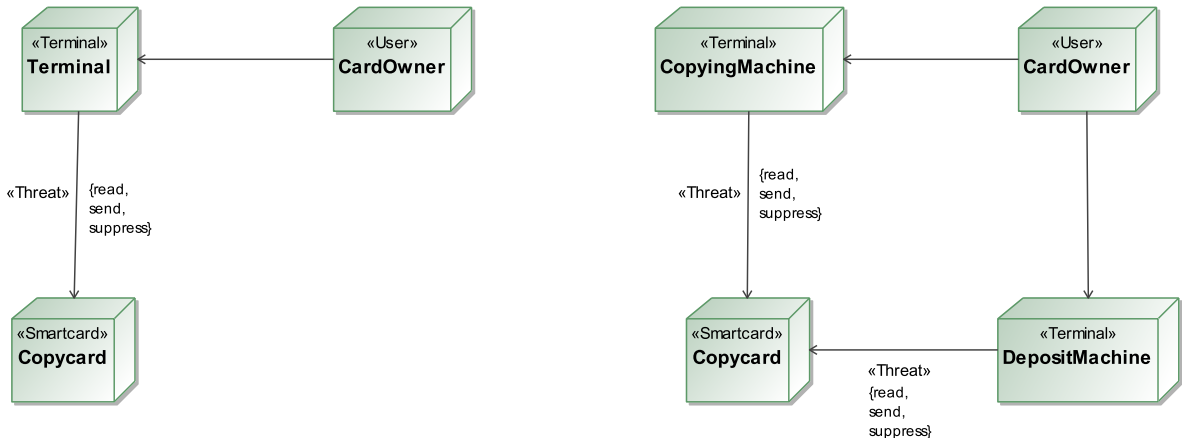

Abbildung 4.17.: Zwei mögliche Deploymentdiagramme für die Kopierkartenanwendung

Zusätzlich zur Kommunikationsstruktur sind in den Deploymentdiagrammen die Fähigkeiten eines möglichen Angreifers modelliert. Im Beispiel der Kopierkartenanwendung wird angenommen, dass ein Angreifer keine Möglichkeit hat, die Nachrichten, die der Benutzer `CardOwner` sendet oder empfängt, zu lesen, zu unterdrücken oder über diesen Kommunikationspfad eigene Nachrichten zu senden. Der Kommunikationspfad ist deshalb nicht mit dem Stereotyp ≪Threat≫ annotiert. Die Kommunikation zwischen einem Terminal und der Kopierkarte kann von einem Angreifer abgehört und unterdrückt werden. Außerdem hat der Angreifer die Fähigkeit, über diese Kommunikationskanäle eigene Nachrichten einzuschleusen. Alle Kommunikationspfade zwischen einem Terminal und dem `Copycard` Knoten sind deshalb mit Stereotyp ≪Threat≫ annotiert und dieser hat die Tags `read`, `send` und `suppress`.

4.4. Dynamische Modellierung

Der dynamische Teil einer sicherheitskritischen Anwendung wird mit Sequenz- und Aktivitätsdiagrammen modelliert. Die Sequenzdiagramme enthalten die zwischen den Kommunikationspartnern ausgetauschten Nachrichten, gleichen also im Wesentlichen der auch in der Literatur verwendeten Beschreibung kryptographischer Protokolle. Diese enthält jedoch nicht alle Details der Protokolle. Insbesondere das Verarbeiten einer Nachricht durch eine Komponente (inklusive der damit verbundenen Zustandsänderung) und das Verhalten im Fehlerfall ist nicht in der Modellierung mit Sequenzdiagrammen enthalten. Die Aktivitätsdiagramme dagegen enthalten die komplette dynamische Sicht der Anwendung. Idee ist es, beim Design der Protokolle mit den Sequenzdiagrammen zu beginnen und diese dann Schritt für Schritt zu verfeinern. Da die Sequenzdiagramme eine vereinfachte Sicht darstellen, ist es somit für den Entwickler leichter, sich zunächst nur die Nachrichten eines Protokolls zu überlegen und sich im Anschluss Gedanken über das vollständige Protokoll zu machen.

Dieser Abschnitt erläutert die dynamische Modellierung einer Anwendung. Abschnitt 4.4.1 beschreibt die Modellierung mit Sequenzdiagrammen. Abschnitt 4.4.2 erläutert die Modellierung mit Aktivitätsdiagrammen und geht dabei auch auf die Verwendung der Sprache MEL in den Aktivitätsdiagrammen ein. Wie auch bei der statischen Modellierung, gibt es für die dynamische Modellierung einige Richtlinien und Einschränkungen. Diese sind notwendig, um eine fehlerfreie Transformation der Modelle zu ermöglichen und gleichzeitig die Verfeinerungsbeziehung zwischen dem Code und dem formalen Modell sicherzustellen. Die Einhaltung der Richtlinien und Einschränkungen wird während der Validierung, die vor der Ausführung der Transformationen stattfindet, überprüft.

4.4.1. Sequenzdiagramme

4.4.1.1. Modellierung der Kommunikationspartner

Jedes Protokoll wird in einem separaten Sequenzdiagramm modelliert. Für jede an dem Protokoll beteiligte Komponente(ninstanz) enthält das Sequenzdiagramm eine Lebenslinie (`Lifeline`). Mögliche Komponenten sind die im Klassendiagramm definierten Terminal- und Smart Card-Komponenten sowie die dort definierten Benutzer. Die Zuordnung der Lebenslinie zu einer im Klassendiagramm definierten Komponentenklasse erfolgt durch Angabe

eines Typs für die Lebenslinie, für den eine im Klassendiagramm definierte Komponentenklasse gewählt werden muss. Die Angabe eines Instanznamens für die Lebenslinie ist optional.

Abbildung 4.18 zeigt die Komponentenklassen einer möglichen Anwendung (links) sowie die Lebenslinien eines möglichen Sequenzdiagramms (rechts).

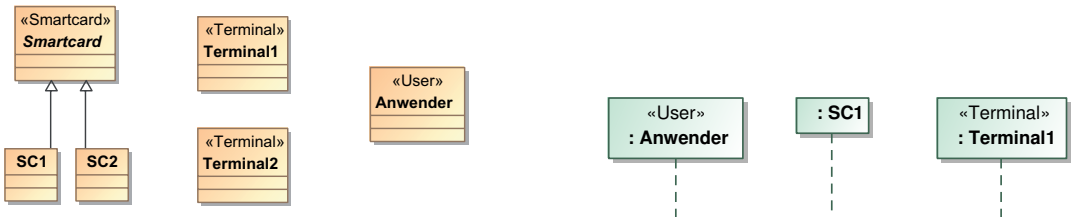

Abbildung 4.18.: Komponentenklassen einer Anwendung sowie die Lebenslinien eines Sequenzdiagramms

In dem Sequenzdiagramm gibt es Lebenslinien für Instanzen der Benutzerklasse `Anwender`, der Smart Card-Komponente `SC1` sowie der Terminalkomponente `Terminal1`.

Für die Typangabe einer Lebenslinie kann auch der Name der abstrakten Oberklasse angegeben werden. Ist eine abstrakte Oberklasse angegeben, ist das Protokoll für alle Subklassen definiert. Dies ist gleichbedeutend damit, dass das Sequenzdiagramm entsprechend der Anzahl der Subklassen dupliziert und für jede Subklasse einzeln definiert wird (durch Modellieren einer Lebenslinie für die konkrete Subklassse statt der Oberklasse).

4.4.1.2. Modellierung der ausgetauschten Nachrichten

Neben den Kommunikationsteilnehmern definiert das Sequenzdiagramm die Nachrichten, die während eines Protokolllaufs zwischen den Teilnehmern ausgetauscht werden. Der Name einer Nachricht muss einem im Klassendiagramm definierten Nachrichtentyp entsprechen. Benutzernachrichten können nur zwischen einer Komponenteninstanz mit Stereotyp ≪User≫ und einer Terminalinstanz ausgetauscht werden. Nachrichten können nur zwischen einer Terminal- und einer Smart Card-Instanz ausgetauscht werden. Ein Protokoll beginnt immer bei einer Benutzerinstanz, d.h. die erste Nachricht wird von einem Benutzer an ein Terminal gesendet. Ein Terminal kann nur nach Empfangen einer Nachricht eine (andere) Nachricht versenden. Eine Smart Card-Komponente kann ebenfalls keine Kommunikation initiieren, sondern nur auf eine empfangene Nachricht antworten. Diese Antwortnachricht muss an die sendende Terminalinstanz zurückgeschickt werden. Dies modelliert das Kommunikationsverhalten von Smart Card-Anwendungen, bei denen die Smart Card nur mit dem Terminal, in dessen Kartenleser sie sich gerade befindet, kommunizieren kann. Eine direkte Kommunikation mit einem Benutzer ist nicht möglich.

Die Argumente der Nachrichten werden in den Sequenzdiagrammen nicht angegeben. Würde man die Argumente angeben wollen, müsste man für jedes der Argumente erläutern wie es definiert ist. Dies ist bei einem komplexen Datum sehr unübersichtlich. Da diese Informationen auch in den Aktivitätsdiagrammen enthalten sind, wird nur der Name der Nachricht angegeben. Über Kommentare kann dann noch detaillierter definiert werden, was das Ziel

der Nachricht ist, zum Beispiel „Anforderung einer neu generierten Nonce“. Die Kommentare enthalten natürliche Sprache und dienen nur dem Verständnis.

Abbildung 4.19 zeigt das Deploymentdiagramm (links) sowie ein mögliches Sequenzdiagramm (rechts) einer Anwendung.

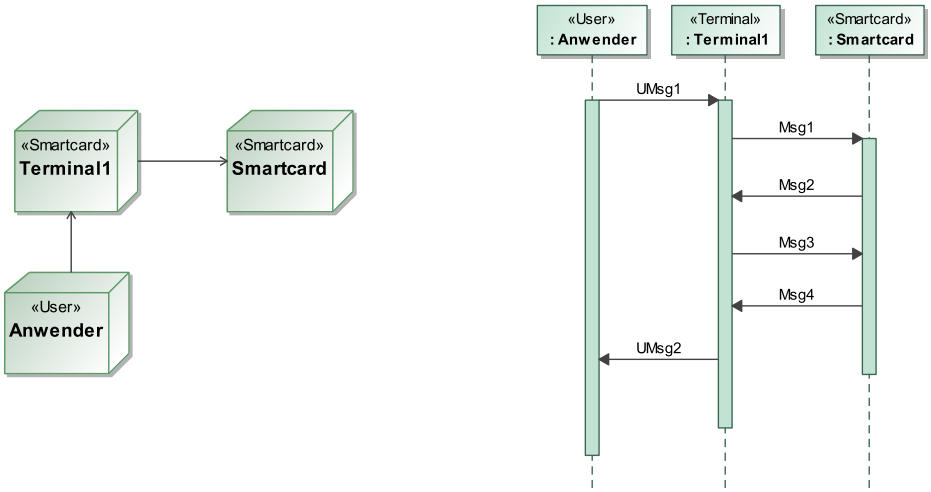

Abbildung 4.19.: Deploymentdiagramm und zugehöriges Sequenzdiagramm einer Anwendung

Die Kommunikation des im Sequenzdiagramm modellierten Protokolls beginnt beim Benutzer (`Anwender`). Die Terminalkomponente darf mit dem Benutzer sowie der Smart Card-Komponente kommunizieren. Eine Nachricht darf jedoch nur nach Empfang einer anderen Nachricht, als Reaktion auf diese, verschickt werden. Die Smart Card-Komponente kann nur mit dem Terminal kommunizieren und auf empfangene Nachrichten antworten. Die Nachrichten `UMsg1` und `UMsg2` sind im Klassendiagramm definierte Nachrichtenklassen mit Stereotyp ≪Usermessage≫, die anderen Nachrichten (`Msg1`-`Msg4`) sind im Klassendiagramm definierte Nachrichtenklassen mit Stereotyp ≪Message≫.

4.4.2. Aktivitätsdiagramme

Die Sequenzdiagramme zeigen nur einige Aspekte der dynamischen Sicht einer Anwendung. Modelliert sind dort lediglich die miteinander kommunizierenden Komponenten sowie die Nachrichten, die diese austauschen. Das Verarbeiten einer Nachricht, d.h. die eigentlichen Protokollschritte, sind nicht dargestellt. Weiterhin ist in den Sequenzdiagrammen nur ein möglicher Ablauf des Protokolls und zwar der Ablauf im positiven Fall, d.h. wenn keine Fehler auftreten, beschrieben. Außerdem sind die konkreten Daten, die mit den Nachrichten ausgetauscht werden, nicht mitmodelliert. Es bleibt damit offen, welche Daten tatsächlich mit einer Nachricht verschickt werden. All diese Details sind in den Aktivitätsdiagrammen enthalten. Sie beschreiben die vollständige dynamische Sicht der modellierten Anwendung und werden als Quelle für die Codegenerierung sowie die Generierung der formalen Spezifikation verwendet.

Dieser Abschnitt beschreibt die Modellierung eines Protokolls mit Aktivitätsdiagrammen. Wie auch bei der Modellierung der statischen Sicht unterstützt der SecureMDD-Ansatz nicht die Verwendung aller für Aktivitätsdiagramme zur Verfügung stehenden UML-Elemente. Zu-

nächst werden deshalb die für die Modellierung genutzten und bei den Transformationen berücksichtigten Elemente beschrieben. Im Anschluss daran wird die Verwendung der Model Extension Language in den Aktivitätsdiagrammen erläutert.

4.4.2.1. Modellierung von Protokollen mit Aktivitätsdiagrammen

Für jedes Protokoll wird ein eigenes Aktivitätsdiagramm erstellt. Ein Aktivitätsdiagramm besteht aus mehreren Aktivitätsbereichen (`ActivityPartitions`). Aktivitätsbereiche werden verwendet, um das Diagramm in Verantwortungsbereiche für die einzelnen Komponenten zu unterteilen. Jede Komponente, die an dem Protokoll beteiligt ist, wird durch einen Aktivitätsbereich repräsentiert. Für jede im zugehörigen Sequenzdiagramm definierte Lebenslinie enthält das Aktivitätsdiagramm einen Aktivitätsbereich. In diesem modelliert sind die „Aktivitäten“ einer Komponente, d.h. das Versenden und Empfangen von Nachrichten sowie das Verarbeiten einer empfangenen Nachricht.

Die Zuordnung einer Komponente zu einem Aktivitätsbereich erfolgt über die Angabe des Namens der Komponentenklasse. Hier kann, wie auch bei den Sequenzdiagrammen, der Name einer abstrakten Oberklasse oder einer Subklasse angegeben werden. Wird eine Superklasse angegeben, ist der Aktivitätsbereich für jede ihrer Subklassen definiert.

Ein Aktivitätsbereich enthält alle Protokollschritte einer Komponente für das im Diagramm definierte Protokoll. Der Hauptteil eines Protokollschritts besteht im Verarbeiten der empfangenen Nachricht, d.h. verschiedenen Überprüfungen (der Daten der empfangenen Nachricht sowie des aktuellen Zustands der Komponente) und Änderungen am Zustand der gerade aktiven Komponente. Ein Protokollschritt endet entweder mit einem Abbruch im Fehlerfall, dem Ende eines Protokolls oder dem Senden einer Nachricht an eine andere am Protokoll beteiligte Komponente.

Der Aktivitätsbereich darf verschiedene UML-Elemente enthalten. Diese Elemente sowie ihre Verwendung sind im Folgenden erläutert und in Abbildung 4.20 dargestellt.

- Ein `ControlFlow` verbindet zwei Modellierungselemente miteinander und beschreibt den Übergang von einem Element zu einem anderen. Durch die Verbindung der Elemente durch `ControlFlows` wird automatisch die Reihenfolge der Abarbeitung der Elemente festgelegt.
- Ein `InitialNode` definiert den Anfangspunkt eines Protokolls. Ein Protokoll wird immer von einem Benutzer angestoßen. Jedes Aktivitätsdiagramm muss genau einen Initialknoten enthalten, der sich in der Partition eines Benutzers, d.h. einer mit ≪User≫ annotierten Klasse, befindet. Dieser hat genau einen ausgehenden `ControlFlow`.
- Ein `FlowFinalNode` beendet einen Protokollschritt und modelliert das Auftreten eines Fehlers. Da zu jeder Zeit immer max. ein Protokollschritt ausgeführt wird, beendet dieser auch die Ausführung des gesamten Protokolls. Ein `FlowFinalNode` hat genau eine eingehende Kante `ControlFlow`.
- Ein `ActivityFinalNode` beendet ebenfalls den aktuellen Protokollschritt und modelliert das erfolgreiche Ende eines Protokolls. Bei Erreichen eines `ActivityFinalNodes` wurde das Protokoll erfolgreich durchlaufen. Ein `ActivityFinalNode` hat genau eine eingehende Kante. Jedes Aktivitätsdiagramm muss genau einen `ActivityFinalNode` enthalten.

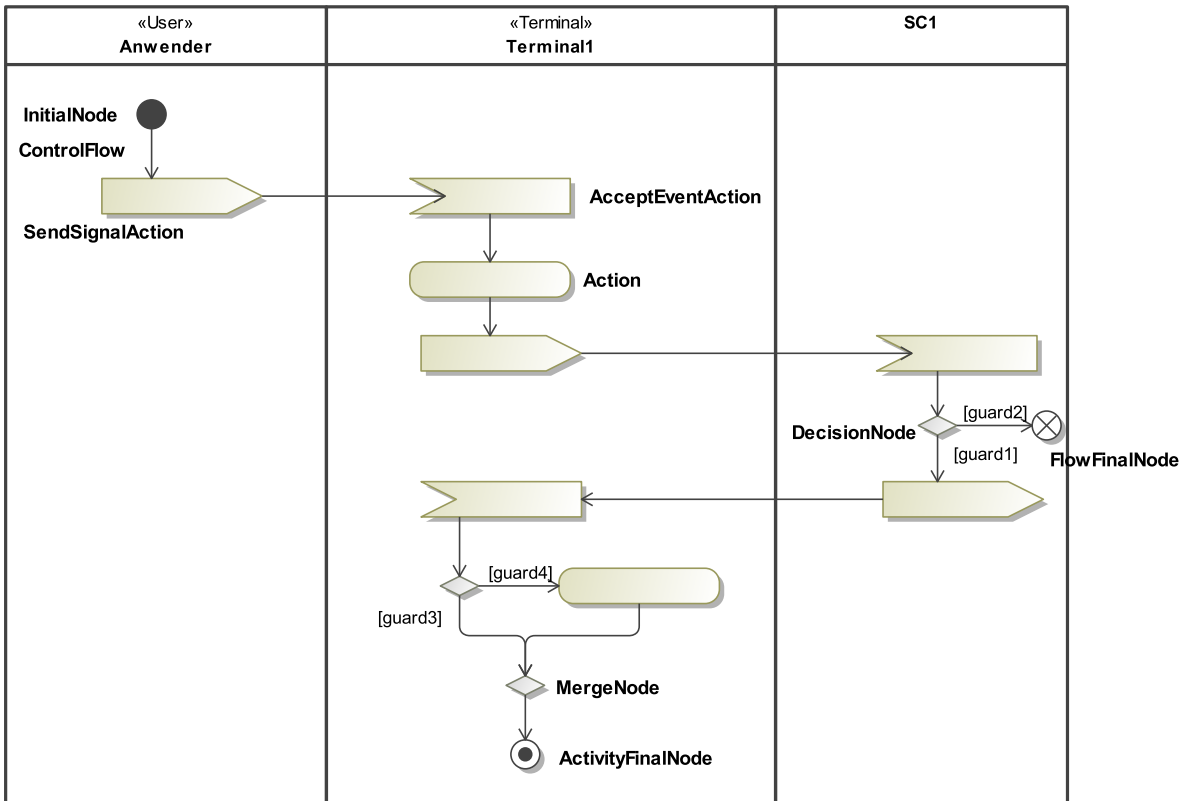

Abbildung 4.20.: Modellierung eines Protokolls in einem Aktivitätsdiagramm (ohne Ausdrücke der Model Extension Language)

- Eine `SendSignalAction` modelliert das Senden einer Nachricht an eine andere Komponente. Sie hat genau einen eingehenden und einen ausgehenden `ControlFlow`. Der eingehende `ControlFlow` befindet sich im selben Aktivitätsbereich wie die `SendSignalAction`. Der ausgehende `ControlFlow` zeigt auf eine `AcceptEventAction`, die sich in einem anderen Aktivitätsbereich als die `SendSignalAction` befindet. Eine `SendSignalAction` beendet einen Protokollschritt. Der nächste Protokollschritt wird von der Komponente, die die nachfolgende `AcceptEventAction` enthält, ausgeführt.
- Eine `AcceptEventAction` modelliert das Empfangen einer Nachricht und gibt an, dass ein neuer Protokollschritt begonnen wird. Eine `AcceptEventAction` hat genau einen eingehenden und einen ausgehenden `ControlFlow`. Der eingehende `ControlFlow` muss bei einer `SendSignalAction` in einem anderen Aktivitätsbereich beginnen.
- Eine `Action` modelliert die Änderung des Zustands der Komponente, in dessen Aktivitätsbereich sie sich befindet. Eine `Action` hat einen eingehenden und einen ausgehenden `ControlFlow`.
- Ein `DecisionNode` modelliert eine Verzweigung und besitzt genau einen eingehenden sowie zwei ausgehende `ControlFlows`. Jeder ausgehender `ControlFlow` hat einen `Guard`. Dieser gibt an, unter welcher Bedingung (abhängig vom aktuellen Zustand

der Komponente und der Belegung der lokalen Variablen) verzweigt wird (siehe auch Abschnitt 4.4.2.2).

- Ein `MergeNode` führt verschiedene Kontrollflüsse wieder zusammen. Ein `MergeNode` hat mindestens zwei eingehende und genau einen ausgehenden `ControlFlow`.

4.4.2.1.1. Verwendung von Subdiagrammen Zusätzlich zu der Beschreibung eines Protokolls werden Aktivitätsdiagramme in SecureMDD dafür verwendet, Teile von Protokollbeschreibungen auszulagern. Dies ist dann sinnvoll, wenn ein Teilablauf in mehreren Protokollen vorkommt. Auf diese Weise müssen Teilfunktionalitäten nur einmal modelliert werden und können dann von verschiedenen Aktivitätsdiagrammen aus aufgerufen werden. Die Verwendung von Subdiagrammen ist vergleichbar mit dem Auslagern von Funktionalität in eine separate Methode auf Programmiersprachenebene.

Ein Subdiagramm bezieht sich immer auf genau eine Komponente und wird innerhalb eines Protokollschritts dieser Komponente aufgerufen. Das Subdiagramm darf keine Kommunikation mit anderen Komponenten enthalten, d.h. das Diagramm enthält nur einen Aktivitätsbereich.

Abbildung 4.21 zeigt exemplarisch zwei mögliche Subdiagramme für die Komponentenklasse `Terminal1`. Ausdrücke der Model Extension Language sind in den Diagrammen nicht enthalten. Das linke Subdiagramm besitzt zwei Eingabe- und einen Ausgabeparameter. Das rechte Diagramm besitzt keine Parameter.

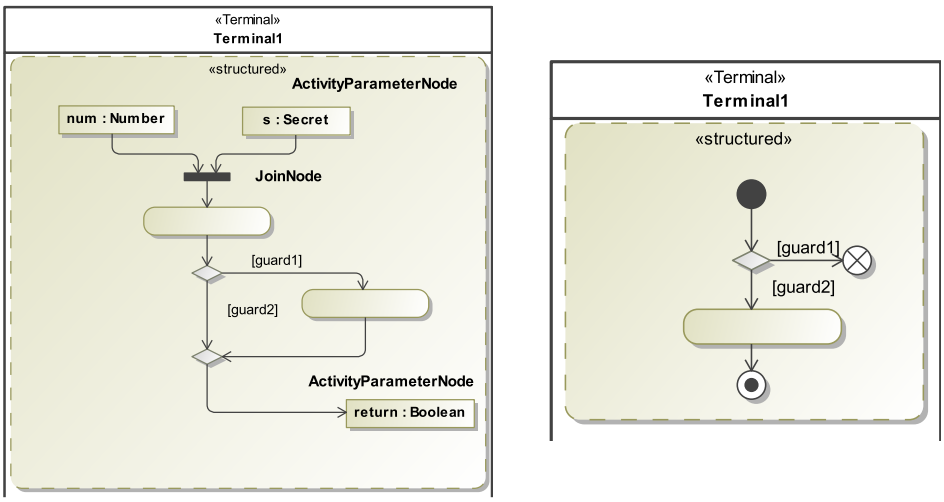

Abbildung 4.21.: Zwei mögliche Subdiagramme einer Anwendung

Ein Subdiagramm ist in einem Aktivitätsdiagramm definiert und enthält einen Aktivitätsbereich für die Komponente, die das Subdiagramm aufrufen kann. Der Aktivitätsbereich enthält einen `StructuredActivityNode`. Der `StructuredActivityNode` enthält die eigentliche Beschreibung der Funktionalität. Einem Subdiagramm können beim Aufruf Argumente übergeben werden, die dann innerhalb des Subdiagramms verwendet werden können. Weiterhin kann ein Subdiagramm einen Rückgabewert haben, der nach Beendigung des Subdiagrammdurchlaufs an die aufrufende Komponente übergeben wird. Die Definition der entsprechenden (Ein- und Ausgabe-) Parameter erfolgt über `ActivityParameterNodes`.

Ein Subdiagramm ohne Eingabeparameter beginnt, wie die Beschreibung eines Protokolls, mit einem `InitialNode` (Abb. 4.21 links). Ein Subdiagramm, das beim Aufruf Argumente übergeben bekommt, definiert für jeden Eingabeparameter einen `ActivityParameterNode` (Abb. 4.21 rechts). Der Kontrollfluss eines Subdiagramms beginnt demnach entweder mit einem `InitialNode` oder der Definition von (mehreren) Eingabeparametern, die mittels eines `JoinNodes` zu einem Kontrollfluss zusammengeführt werden. Ein Subdiagramm endet mit der Angabe eines `ActivityFinalNodes` (für den Fall, dass es keinen Rückgabewert gibt) oder der Angabe eines Ausgabeparameters. Dieser ist ebenfalls als `ActivityParameterNode` definiert. Vor Beenden des Subdiagrammdurchlaufs müssen alle Kontrollflüsse mittels eines `MergeNodes` zu einem Kontrollfluss zusammengefügt werden. Der Parameter des `ActivityParameter Nodes` muss einen Namen sowie einen Typ besitzen. Der Typ muss entweder in den vordefinierten Sicherheitsdatentypen oder als Datenklasse im Klassendiagramm der Anwendung definiert sein. Der Name ist frei wählbar.

Die Modellierung der Subdiagramme erfolgt äquivalent zu denen der Aktivitätsdiagramme zur Protokollbeschreibung, die geltenden Validierungsregeln sind dieselben. Es dürfen die UML-Elemente `Action`, `ControlFlow`, `DecisionNode`, `FlowFinalNode` und `MergeNode` verwendet werden. Die Verwendung von `SendSignalActions` und `AcceptEventActions`, d.h. das Versenden von Nachrichten an eine andere Komponente, ist nicht erlaubt.

Der Aufruf eines Subdiagramms erfolgt in einer `Action` und wird mit der MEL modelliert. Dies ist in Abschnitt 4.4.2.2 erläutert. Die `Action`, in der das Subdiagramm aufgerufen wird, muss sich im Aktivitätsbereich der Komponente befinden, für die das Subdiagramm definiert ist. Bei Aufruf des Subdiagramms wartet die aufrufende Komponente bis der Subdiagrammdurchlauf abgeschlossen ist. Dies ist der Fall, wenn im Subdiagramm der `ActivityFinalNode` (für ein Subdiagramm ohne Ausgabeparameter) oder der (Ausgabe-)`ActivityParameterNode` erreicht wird. In diesen Fällen wird der Protokollschritt, in dem das Subdiagramm aufgerufen wurde, nach erfolgreichem Durchlauf desselben fortgesetzt. Außerdem ist ein Subdiagrammdurchlauf beendet, wenn während des Abarbeitens des Subdiagramms ein Fehler auftritt. Dies ist durch einen `FlowFinalNode` modelliert. In diesem Fall ist das Verhalten dasselbe, als wenn im Protokollschritt der aufrufenden Komponente ein Fehler (modelliert durch einen `FlowFinalNode`) auftritt. Der Fehler wird somit an die aufrufende Komponente weitergegeben und der aktuelle Protokollschritt wird beendet.

4.4.2.2. Verwendung der Model Extension Language in Aktivitätsdiagrammen

Die Modellierung mit Aktivitätsdiagrammen reicht nicht aus, um die dynamische Sicht einer Anwendung vollständig zu modellieren. Es fehlt eine textuelle Sprache, die die verwendeten UML-Elemente ergänzt und die Modellierung zu einem so hohen Detaillierungsgrad unterstützt, dass die vollautomatische Generierung von Code und einer formalen Spezifikation möglich ist. Zu diesem Zweck definiert und verwendet der SecureMDD-Ansatz die Model Extension Language (MEL), die auf die Modellierung von kryptographischen Protokollen zugeschnitten ist. Die Sprache ist eng mit den UML-Aktivitätsdiagrammen verzahnt und stellt eine Erweiterung dieser dar. In diesem Abschnitt wird deshalb die Verwendung von MEL in Aktivitätsdiagrammen erläutert. Die vollständige Syntax und Semantik der Sprache sind in Kapitel 5 definiert.

4.4.2.2.1. MEL in Actions Innerhalb einer UML-`Action` dürfen beliebig viele MEL-Ausdrücke $expression_1$; $expression_2$; ... stehen. Diese müssen durch Semikolon voneinander getrennt sein. Die Reihenfolge der Ausdrücke gibt die Reihenfolge der Auswertung an, d.h. es wird sequentiell zunächst $expression_1$ ausgewertet, dann $expression_2$ usw.. Anstatt mehrere Ausdrücke in einer `Action` zu modellieren, ist es analog auch möglich, für jeden Ausdruck eine eigene `Action` zu verwenden und diese, verbunden durch `ControlFlow`-Elemente, hintereinanderzuschreiben.

4.4.2.2.2. MEL in Guards von Fallunterscheidungen Die ausgehenden Kanten einer bedingten Verzweigung (modelliert durch einen `DecisionNode`) definieren zwei alternative Kontrollflüsse. Die Guards der `ControlFlows` geben an, welche Bedingung erfüllt sein muss, damit der entsprechende Kontrollfluss durchlaufen wird. Die Guards müssen jeweils einen Ausdruck der Sprache MEL enthalten, der zu einem booleschen Wert ausgewertet wird.

Abbildung 4.22 zeigt die Verwendung von MEL innerhalb von Guards (für `ControlFlow`-Elemente) zur Modellierung von bedingten Verzweigungen.

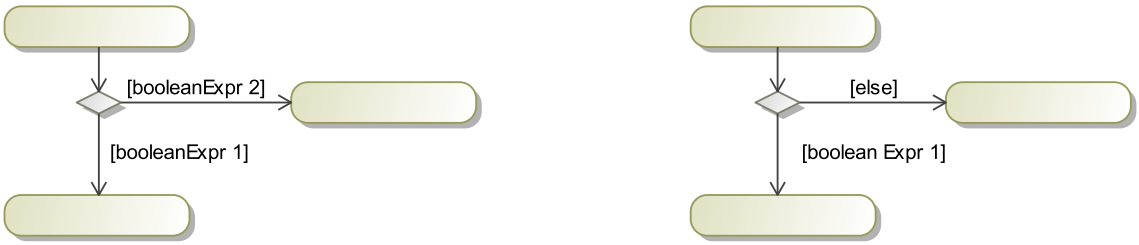

Abbildung 4.22.: Verwendung von MEL innerhalb von Guards.

Der Ausdruck *booleanExpr*1 muss ein MEL-Ausdruck sein, der zu einem booleschen Wert ausgewertet wird. Der Ausdruck *booleanExpr*2 muss ebenfalls zu einem booleschen Wert und zwar zu den komplementären Wert von *booleanExpr*1 ausgewertet werden. Zur Vereinfachung ist für einen der beiden Guards auch die Angabe des MEL-Schlüsselwortes `else` zulässig. Dies bedeutet, dass der so gekennzeichnete Guard den negierten Ausdruck des anderen Guards enthält. Es muss für beide ausgehende Kanten eines `DecisionNodes` Guards angegeben sein. Die Nutzung von Guards ist in SecureMDD nur an den ausgehenden Kanten von `DecisionNodes` erlaubt.

4.4.2.2.3. Verwendung der im Klassendiagramm definierten Datentypen in Aktivitätsdiagrammen Innerhalb eines MEL-Ausdrucks kann auf die in den UML-Klassendiagrammen definierten Datenklassen, Methoden und Felder einer Komponente zugegriffen werden. Beispielsweise kann innerhalb der Aktivitätsdiagramme durch Angabe des MEL-Schlüsselwortes *create* ein Objekt einer im Klassendiagramm definierten Klasse erzeugt werden, dort deklarierte Methoden aufgerufen werden oder auf die Daten einer Komponente (Attribute und Assoziationsenden) zugegriffen werden. Innerhalb eines Aktivitätsbereichs sind genau die Klassen und Methoden sichtbar (und können verwendet bzw. aufgerufen werden), die von der zu dem

Aktivitätsbereich gehörenden Komponentenklasse erreichbar sind. Erreichbar sind alle Klassen, zu denen die Komponentenklasse Assoziationen und Usage-Dependencies besitzt. Hierzu zählen auch Klassen, die transitiv, über weitere Klassen hinweg, erreichbar sind.

Für die Erzeugung von kryptographischen Daten wie Nonces, Schlüsseln sowie für die Durchführung kryptographischer Operationen sind in MEL entsprechende Operationen vordefiniert (siehe Abschnitt 5.4).

4.4.2.2.4. Erzeugen von Literalen Bei der Modellierung eines Protokolls in den Aktivitätsdiagrammen ist es manchmal erforderlich, ein Literal (vom Typ `Boolean`, `Number` oder `String`) zu verwenden und dieses z.B. einer Variablen zuzuweisen. Literale vom Typ `Boolean` werden mit den Schlüsselwörtern *true* und *false* erzeugt. Die Zuweisung des Wertes *true* an die boolesche Variable `b` hat die Syntax „`b := true`“. Literale vom Typ `Number` werden durch Angabe des konkreten Wertes erzeugt. Die Zuweisung des Wertes *10* an die Variable `num` vom Typ `Number` hat die Syntax „`num := 10`“. Die Erzeugung von Stringliteralen erfolgt durch Angabe des Strings, umschlossen von doppelten Anführungszeichen. Die Zuweisung des Stringliterals `String` an die Variable `str` hat die Syntax `str := String`. Stringliterale dürfen nur innerhalb des Aktivitätsbereichs einer Terminal- oder Userkomponente verwendet werden. Für Smart Card-Komponenten ist die Verwendung von Stringliteralen nicht erlaubt.

4.4.2.2.5. Versenden von Nachrichten Das Senden einer Nachricht wird in den Aktivitätsdiagrammen durch eine UML-`SendSignalAction` beschrieben. Der darin enthaltene Ausdruck muss eine MEL-`SendSignalAction` sein. Eine MEL-`SendSignalAction` besteht aus dem Namen eines im Klassendiagramm definierten Nachrichtentyps, von dem ein Objekt erzeugt und versendet werden soll. Die Nachrichtenklasse kann entweder eine ≪Message≫- oder eine ≪Usermessage≫-Klasse sein. Hat die Klasse Attribute oder Assoziationen auf andere Datenklassen, müssen hierfür zusätzlich Argumente in der Form „`name(arg1,arg2,..)`“ angegeben werden. Als Argumente können die Namen der Attribute oder Assoziationsenden der aktuellen Komponente oder lokal gültige Variablen vom richtigen Typ angegeben werden. Die Reihenfolge der Argumente eines Nachrichtenobjekts ergibt sich aus dem Klassendiagramm. Hier gilt die Regel, dass zuerst die Attribute der Klasse (in der Reihenfolge ihrer Definition) und dann die Assoziation genannt werden. Hat eine Klasse mehr als eine ausgehende Assoziation, muss für die Klasse ein Konstruktor definiert sein (siehe Abschnitt 4.2.1.7). Dieser gibt die Reihenfolge der Parameter vor.

Gibt es zwischen zwei Komponentenklassen mehrere Kommunikationskanäle (im Deploymentdiagramm, siehe Abschnitt 4.2.2.1), muss beim Senden der Nachricht zusätzlich angegeben werden, über welchen der Kanäle die Nachricht gesendet werden soll. Zu diesem Zweck haben die Enden der Kommunikationskanäle einen eindeutigen Namen. Diese Enden werden in SecureMDD Ports genannt. Eine MEL-`SendSignalAction` hat in diesem Fall die Form „name(arg1,arg2,..) via portname“. `via` ist ein in MEL definiertes Schlüsselwort, dem die Angabe eines Portnamens `portname` folgt. Es dürfen nur Portnamen verwendet werden, die im Deploymentdiagramm definiert sind.

Um das Versenden einer Nachricht zu illustrieren, sind in Abbildung 4.23 drei mögliche Nachrichtenklassen dargestellt. Entsprechende MEL-`SendSignalActions` zum Versenden von Instanzen dieser Nachrichtenklassen haben folgende Form:

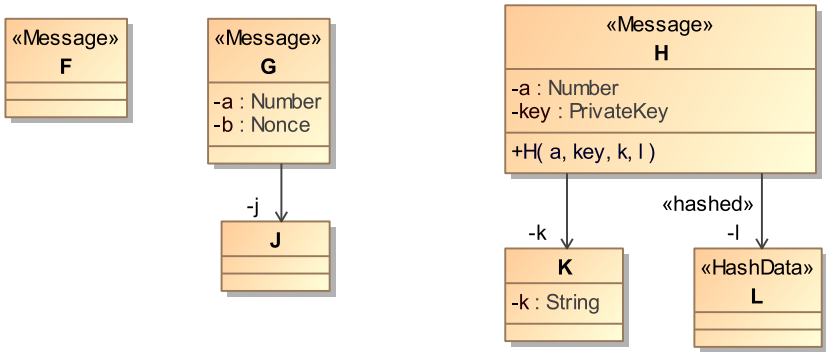

Abbildung 4.23.: Mögliche Nachrichtenklassen einer Anwendung

- `F` für das Versenden eines Nachrichtenobjekts vom Typ `F` ohne Argumente
- `G(arg_a, arg_b, arg_j)` für das Versenden eines Nachrichtenobjekts vom Typ `G` mit den Argumenten `arg_a` (vom Typ `Number`), `arg_b` (vom Typ `Nonce`) und `arg_j` (vom Typ `J`).
- `H(arg_a, arg_key, arg_k, arg_l)` für das Versenden eines Nachrichtenobjekts vom Typ `H` mit den Argumenten `arg_a` (vom Typ `Number`), `arg_key` (vom Typ `PrivateKey`), `arg_k` (vom Typ `K`) und `arg_l` (vom Typ `HashedData`). Die Reihenfolge der Argumente ist in dem Konstruktor der Klasse definiert.

4.4.2.2.6. Empfang einer Nachricht Das Empfangen einer Nachricht wird in den Aktivitätsdiagrammen durch eine UML-`AcceptEventAction` modelliert, die eine MEL-`AcceptEventAction` enthält. Diese besteht aus dem Namen der empfangenen Nachricht, wenn die zugehörige Klasse keine Attribute und Assoziationen besitzt. Anderenfalls muss zusätzlich für jedes Attribut und Assoziationsende der Nachrichtenklasse ein Name für eine lokale Variable angegeben werden. Für die in der Nachricht enthaltenen Daten, die beim Senden der Nachricht als Argumente übergeben wurden, werden beim Empfang der Nachricht lokale Variablen deklariert. Diesen werden die Werte der entsprechenden ausgewerteten Argumentausdrücke zugewiesen. Der MEL-Ausdruck `G(var_a, var_b, var_j)` in einer `AcceptEventAction` führt dazu, dass die in der empfangenen Nachricht vom Typ `G` enthaltenen Argumente `arg_a`, `arg_b` und `arg_j` in drei neu deklarierten lokalen Variablen `var_a`, `var_b` und `var_j` gespeichert werden. Die lokalen Variablen haben die Typen `Number`, `Nonce` und `J`.

4.4.2.2.7. Aufruf von Methoden MEL definiert eine Reihe von Methoden, zum Beispiel zum Verschlüsseln eines Dokuments oder den Zugriff auf eine Liste. Weiterhin gibt es in SecureMDD die Möglichkeit im Klassendiagramm Operationen zu definieren, die nach der Generierung des Codes von Hand implementiert werden sollen (siehe 4.2.1.8). Beide Arten von Methoden können in den Aktivitätsdiagrammen aufgerufen werden. Methodenaufrufe werden in einer UML-`Action` modelliert und durch einen MEL-`MethodCall` angegeben. Hat eine statische Methode `meth` die Parameter `p1` und `p2`, erfolgt der entsprechende Aufruf durch den Ausdruck `meth(arg_p1,arg_p2)`. `arg_p1` und `arg_p2` sind die übergebenen

Argumente, sie müssen den gleichen Typ wie `p1` bzw. `p2` besitzen. Für eine nicht-statische Methode muss zusätzlich ein MEL-Ausdruck `expr` angegeben werden, der nach Auswertung ein Objekt der Klasse, in der die Methode definiert ist, zurückliefert. In diesem Fall kann die Methode durch den MEL-Ausdruck `expr.meth(arg_p1,arg_p2)` aufgerufen werden.

4.4.2.2.8. Aufruf von Subdiagrammen Eine spezielle Form eines Methodenaufrufs ist der Aufruf eines Subdiagramms. Ein Subdiagrammaufruf findet ebenfalls in einer UML-`Action` statt. Ein Subdiagramm darf nur dann aufgerufen werden, wenn der Aktivitätsbereich, von dem aus das Diagramm aufgerufen wird, und der im Subdiagramm definierte Aktivitätsbereich die gleichen sind.

Ist ein Subdiagramm mit dem Namen `subdiag` definiert, erfolgt der Aufruf durch den Ausdruck `subdiag()`. Besitzt ein Subdiagramm Eingabeparameter, enthält der Subdiagrammaufruf zusätzlich Argumente. Die Modellierung erfolgt analog zu dem Aufruf von Methoden mit Parametern.

Weiterhin können Subdiagramme einen Rückgabewert haben. Der Rückgabewert eines Subdiagramms wird wie der Rückgabewert eines Methodenaufrufs behandelt. Er kann z.B. einer Variablen zugewiesen werden.

4.4.2.2.9. Gültigkeitsbereiche lokaler Variablen und Parameter Lokale Variablen sind ab dem Moment der Deklaration bis zum Ende des Protokollschritts, in dem sie deklariert werden, gültig. Die (Ein- und Ausgabe-)Parameter eines Subdiagramms sind innerhalb des gesamten Subdiagramms gültig. Lokale Variablen, die in einem Subdiagramm definiert werden, sind ab dem Moment der Deklaration und bis zum Ende des Subdiagramms gültig.

4.5. Dynamische Modellierung der Kopierkartenanwendung

In diesem Abschnitt wird die dynamische Modellierung der Kopierkartenanwendung vorgestellt. Abschnitt 4.5.1 beschreibt die Modellierung der Anwendung mit Sequenzdiagrammen, Abschnitt 4.5.2 stellt die Aktivitätsdiagramme vor.

4.5.1. Sequenzdiagramme der Kopierkartenanwendung

Die Kopierkartenanwendung bietet die Möglichkeit den Kontostand der Kopierkarte abzufragen die Karte aufzuladen sowie Kopien anzufertigen und diese mit der Kopierkarte zu bezahlen. Jede Funktionalität ist in einem separaten Sequenzdiagramm modelliert. Diese sind im Folgenden beschrieben.

4.5.1.1. Abfragen des Kontostands

Abbildung 4.24 zeigt das Sequenzdiagramm für das Protokoll zum Abfragen des Kontostands einer Kopierkarte an einem Ladeterminal (1). Der Benutzer sendet eine `URequestBalance` Nachricht an das Ladeterminal. Dieses schickt daraufhin eine Nachricht vom Typ `RequestBalance` an die Kopierkarte (2), die auf diese Nachricht hin eine

ResRequestBalance-Nachricht zurück an das Ladeterminal schickt (3). Diese enthält den aktuellen Kontostand der Karte. Das Ladeterminal schickt diesen Kontostand mittels einer UShowBalance-Nachricht zurück an den Benutzer (4).

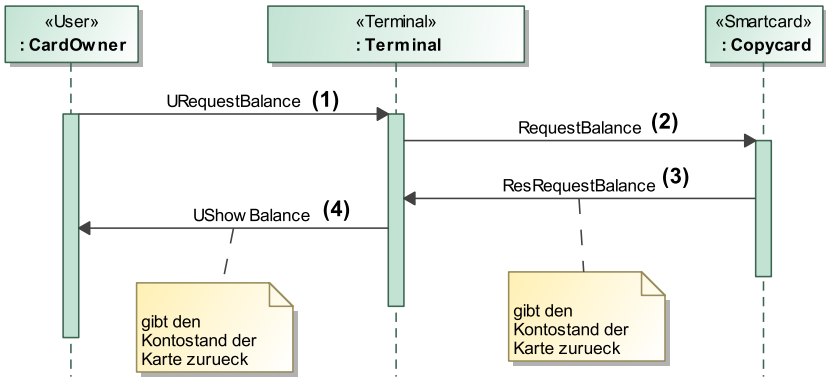

Abbildung 4.24.: Sequenzdiagramm für das Abfragen des Kontostands

4.5.1.2. Aufladen der Kopierkarte

In Abbildung 4.25 ist das Protokoll zum Laden eines Geldbetrags auf eine Kopierkarte dargestellt. Der Benutzer schickt die Nachricht UInsertMoney an das Ladeterminal (1). Diese modelliert das Einwerfen von Geld in das Ladeterminal und signalisiert gleichzeitig, dass die Karte mit dem entsprechenden Betrag aufgeladen werden soll. Der nächste Schritt ist nun die Authentisierung des Ladeterminals gegenüber der Karte. Das Terminal sendet deshalb eine Authenticate-Nachricht an die Kopierkarte (2). Dies ist eine Aufforderung, eine neue Nonce zu generieren. Die Karte antwortet auf diese Nachricht mit einer ResAuthenticate-Nachricht, die die neu generierte Nonce enthält (3). Das Ladeterminal beweist durch Generieren eines Hashwertes bestehend aus den gemeinsamen Geheimnis und der neu generierten Nonce, dass es ein echtes Terminal ist und sendet diesen Hashwert zusammen mit dem zu ladenden Betrag an die Karte zurück (Nachricht Load) (4). Die Karte überprüft den Hashwert und erhöht ihren Kontostand entsprechend dem in der Nachricht enthaltenen Betrag. Als Bestätigung schickt sie eine ResLoad-Nachricht zurück an das Ladeterminal, die den aufgeladenen Betrag enthält (5). Das Ladeterminal bestätigt den erfolgreichen Ladevorgang dem Benutzer CardOwner durch Senden einer Nachricht vom Typ UResLoad (6). Alle verschickten Nachrichten(typen) sind im Klassendiagramm definiert.

4.5.1.3. Bezahlen mit der Kopierkarte

Das Protokoll zum Anfertigen von Kopien und dem Abbuchen des entsprechenden Betrags von der Karte ist in Abbildung 4.26 dargestellt. Beim Bezahlen mit der Kopierkarte ist es für die Sicherheit des Protokolls wichtig, dass das Bezahlen nur mit einer echten Kopierkarte möglich ist. Aus diesem Grund beinhaltet das Protokoll die Authentisierung der Karte gegenüber dem Kopiergerät. Der Benutzer sendet eine URequestCopies-Nachricht an das Kopiergerät (1). Diese Nachricht enthält die Anzahl der anzufertigen Kopien bzw. den entsprechend abzubuchenden Geldbetrag. Das Kopiergerät generiert daraufhin eine neue Nonce

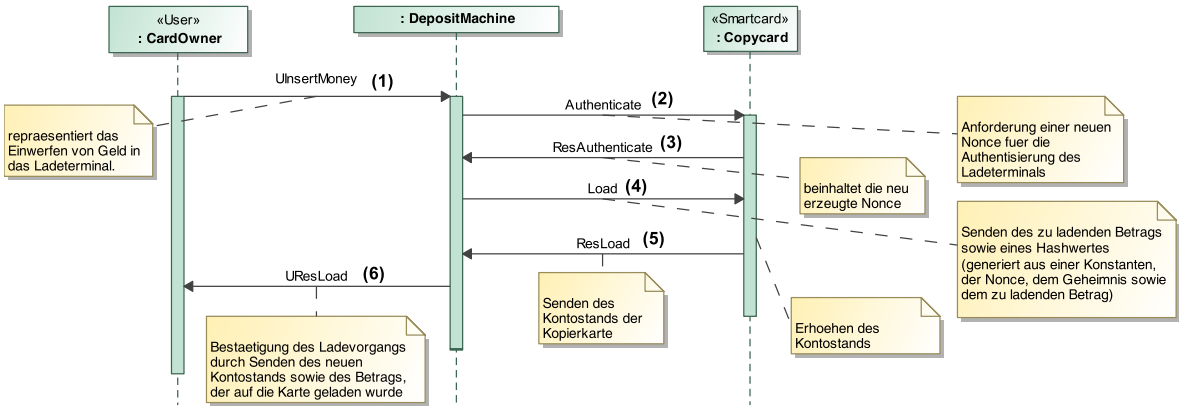

Abbildung 4.25.: Sequenzdiagramm für das Aufladen einer Kopierkarte

und sendet diese, zusammen mit dem Betrag, an die Kopierkarte (2). Die Karte reduziert ihren Kontostand entsprechend des Betrags und bildet einen Hashwert aus dem gemeinsamen Geheimnis, der Nonce sowie dem Betrag und sendet den Hashwert in einer `ResPay`-Nachricht zurück an das Kopiergerät (3). Dieses überprüft den Hashwert. Stimmt er überein, weiß das Kopiergerät, dass die Karte echt ist (da ihr das Geheimnis bekannt ist), die Nachricht neu ist (da die aktuelle Nonce enthalten ist) und der korrekte Betrag von der Karte abgebucht wurde. Das Kopiergerät erlaubt somit das Anfertigen der Kopien und sendet eine `UIssueCopies`-Nachricht zurück an den Benutzer (4).

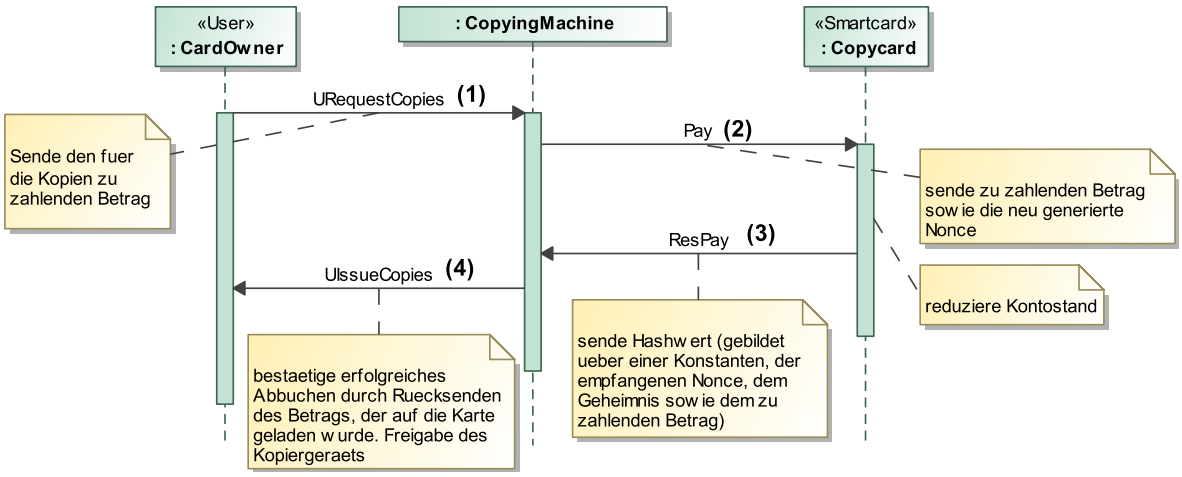

Abbildung 4.26.: Sequenzdiagramm für das Anfertigen von Kopien

4.5.2. Aktivitätsdiagramme der Kopierkartenanwendung

Im Folgenden werden die Aktivitätsdiagramme der Kopierkartenanwendung vorgestellt. Diese drei Diagramme definieren die Protokolle für das Abfragen des Kontostands, das Aufladen einer Kopierkarte sowie das Bezahlen von Kopien an einem Kopiergerät. Die Diagramme

beziehen sich auf die in Abschnitt 4.3 vorgestellten Klassen- und Deploymentdiagramme der Anwendung.

4.5.2.1. Abfragen des Kontostands

Abbildung 4.27 zeigt das Aktivitätsdiagramm, dass das Protokoll zum Abfragen des Kontostands definiert. Für die an dem Protokoll beteiligten Komponentenklassen (`CardOwner`, `Terminal` und `Copycard`) ist jeweils ein Aktivitätsbereich definiert.

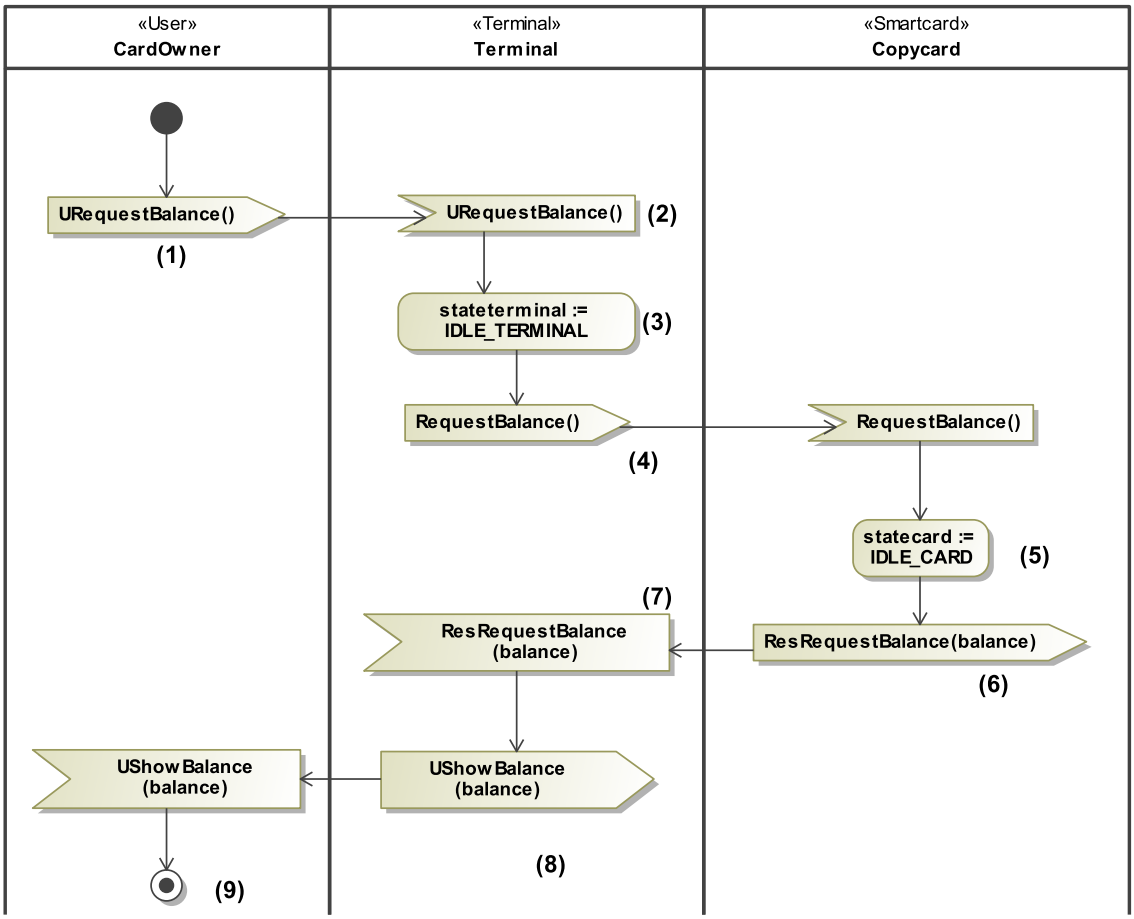

Abbildung 4.27.: Aktivitätsdiagramm für das Abfragen des Kontostands

1. Die Aktivität, und somit das Protokoll zum Abfragen des Kontostands, wird durch den Benutzer gestartet. Der `InitialNode` ist deshalb im Aktivitätsbereich des Benutzers enthalten. Der Benutzer teilt dem Terminal mit, dass er gerne den Kontostand abfragen möchte, d.h. er schickt eine Nachricht vom Typ `URequestBalance` an das Terminal. Das Senden dieser Nachricht ist durch einen `SendSignalNode` modelliert, der ausge-

hende `ControlFlow` führt in die Partition des Terminals. Damit ist der Protokollschritt des Benutzers beendet.

2. Das Terminal empfängt die Nachricht und startet somit den Protokollschritt zum Verarbeiten dieser Nachricht. Das Empfangen ist durch eine UML-`AcceptEventAction` modelliert, die eine MEL-`AcceptEventAction` (mit dem Namen der Nachricht) enthält.
3. Im Anschluss setzt das Terminal seinen Zustand (modelliert durch das Assoziationsende `stateterminal` im Klassendiagramm) auf IDLE_TERMINAL. Dies ist in einer `Action` durch ein MEL-Zuweisung modelliert.
4. Dann sendet das Terminal eine `RequestBalance` Nachricht, die keine Argumente besitzt, an die Kopierkarte. Damit endet dieser Protokollschritt.
5. Die Kopierkarte empfängt die Nachricht. Anschließend setzt sie ihren Zustand `statecard` auf IDLE_CARD.
6. Dann sendet die Kopierkarte eine `ResRequestBalance`-Nachricht zurück an das Terminal. Diese Nachricht enthält den aktuellen Kontostand der Karte, der in dem Attribut `balance` der Karte gespeichert ist. Die `ResRequestBalance`-Nachricht besitzt als Argument das Attribut `balance` bzw. dessen aktuellen Wert, der in der Nachricht übertragen wird.
7. Diese Nachricht wird anschließend vom Terminal empfangen. Der in der entsprechenden `AcceptEventAction` enthaltene MEL-Ausdruck `ResRequestBalance(balance)` bedeutet, dass eine für das Terminal sichtbare lokale Variable namens `balance` deklariert wird, der der in der Nachricht enthaltene Kontostand der Karte zugewiesen wird. Bis zum Ende dieses Protokollschritts (d.h. bis zum Senden der Nachricht `UShowBalance`) kann also auf die Variable `balance` bzw. dessen Wert zugegriffen werden. Würde das Terminal bereits ein Attribut mit diesem Namen besitzen, könnte mittels des MEL-Ausdrucks `self.balance` auf den Wert des Attributs zugegriffen werden.
8. Das Terminal sendet dann eine `UShowBalance`-Nachricht an den Benutzer `CardOwner`. Damit ist der Protokollschritt beendet. Die Nachricht enthält als Argument den zuvor empfangenen Kontostand der Karte, der in der lokalen Variablen `balance` zwischengespeichert ist. An dieser Stelle ist erkennbar, dass die Kommunikation zwischen Terminal und Benutzer auf die gleiche Weise modelliert wird wie die Kommunikation zwischen einem Terminal und einer Smart Card.
9. Der Benutzer empfängt die `UShowBalance`-Nachricht mit dem aktuellen Kontostand (d.h. die Nachricht bzw. die darin enthaltenen Daten werden ihm z.B. über eine Benutzeroberfläche angezeigt). An dieser Stelle ist das Protokoll beendet. Dies ist mit einem `ActivityFinalNode` modelliert.

4.5.2.2. Aufladen der Kopierkarte

Abbildung 4.28 zeigt das Protokoll zum Laden eines Geldbetrags auf eine Kopierkarte. An dem Protokoll sind der Benutzer `CardOwner`, das Ladeterminal `DepositMachine` sowie

die Kopierkarte `CopyingMachine` beteiligt, d.h. es gibt drei Aktivitätsbereiche, die die drei Komponentenklassen repräsentieren.

1. Der Benutzer beginnt das Protokoll, indem er dem Ladeterminal mitteilt, dass er Geld auf die Karte laden möchte. Dies geschieht durch Erzeugen einer Instanz der `UInsertMoney` Nachrichtenklasse und dem anschließenden Senden dieser Instanz an das Ladeterminal. Diese hat ein Argument vom Typ `Number`, das den Wert, der auf die Karte geladen werden soll, enthält. Dieser Wert ist in dem Attribut `value` der Klasse `CardOwner` gespeichert.
2. Das Ladeterminal empfängt die Nachricht vom Typ `UInsertMoney` und speichert den übertragenen Wert in der lokalen Variablen `val`, die an dieser Stelle deklariert wird.
3. Das Ladeterminal prüft dann, ob der Wert dieser Variablen, d.h. der auf die Karte zu ladende Betrag, größer ist als Null. Wäre der Wert kleiner als null, bedeutet dies das Abbuchen von Geld von der Karte und somit ein Fehlerfall. Dies ist modelliert durch eine bedingte Verzweigung mit zwei ausgehenden Kanten (`ControlFlows`). Die Guards der Kanten enthalten jeweils einen MEL-Ausdruck, der zu einem booleschen Wert ausgewertet werden kann. Die Bedingung „`val > 0`“ beschreibt den positiven Fall. Für den negativen Fall („`val <= 0`“) ist das MEL-Schlüsselwort `else` angegeben. Dies bedeutet, dass der Ausdruck immer zu dem zu „`val > 0`“ komplementären Wert ausgewertet wird. Ist `val <= 0`, endet das Protokoll mit einem `FlowFinalNode`. Dieser steht für das Auftreten eines Fehlers. Der Protokollschritt wird in diesem Fall abgebrochen, die lokalen Variablen sind nicht mehr gültig. Ist die Bedingung erfüllt, wird der andere Zweig ausgeführt.
4. Die als nächstes betrachtete `Action` enthält die MEL-Zuweisung `amountToLoad := val;`. Dort wird dem Attribut `amountToLoad` des Ladeterminals der Wert der lokalen Variablen `val` zugewiesen. Im Anschluss daran wird der Zustand `stateterminal` des Ladeterminals auf den Wert EXPRESAUTH gesetzt. Dies steht für „expecting a ResAuthenticate message“ und gibt an, dass das Terminal bei einem fehlerfreien Protokolllauf als nächstes eine eingehende `ResAuthenticate`-Nachricht erwartet.
5. Das Ladeterminal startet nun das eigentliche Protokoll zum Aufladen der Karte und schickt eine `Authenticate`-Nachricht an die Kopierkarte. Wie in Abschnitt 3.3.1 erläutert, beginnt das Protokoll mit der Authentisierung des Terminals. Die `SendSignalAction` enthält den Namen der Nachricht, der ausgehende `ControlFlow` führt in den Aktivitätsbereich der Kopierkarte. Somit ist der Protokollschritt des Ladeterminals beendet und der nächste Schritt startet auf Seite der Kopierkarte.
6. Die Kopierkarte empfängt die Nachricht `Authenticate`.
7. In diesem Protokollschritt generiert die Kopierkarte eine neue Nonce für das *Challenge-Response Verfahren* zwischen Karte und Ladeterminal. Diese Nonce wird im Attribut `challenge` gespeichert. Für das Generieren von Nonces stellt MEL eine vordefinierte Methode zur Verfügung, die innerhalb einer Action verwendet werden kann. Der Ausdruck `challenge := generateNonce()` generiert somit eine neue Nonce und weist sie dem Attribut `challenge` zu.
8. Dann setzt die Karte ihren Zustand `statecard` auf EXPLOAD. Dies steht für „expecting a Load message“ und gibt an, dass die Karte als nächste eingehende

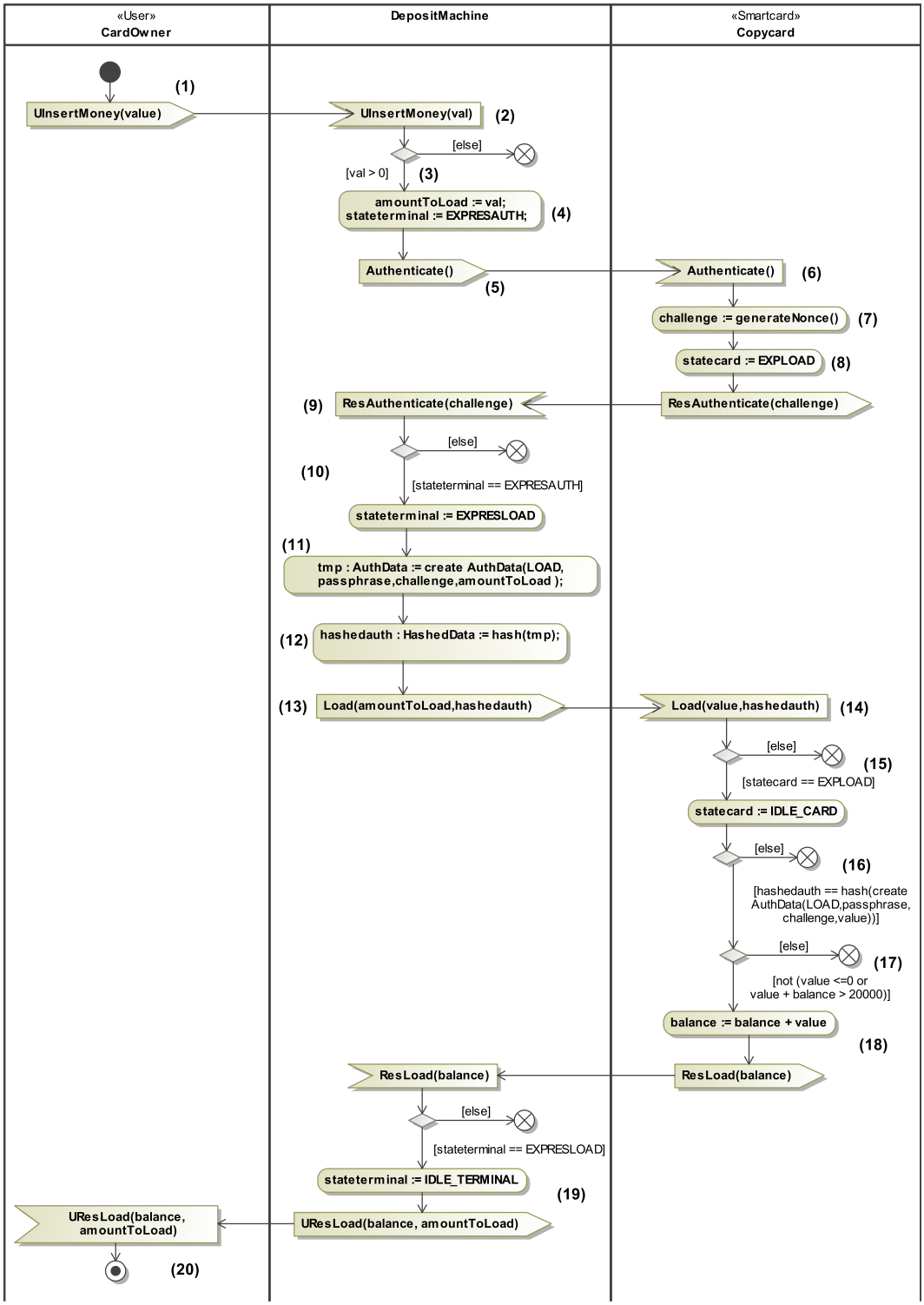

Abbildung 4.28.: Aktivitätsdiagramm für das Aufladen der Kopierkarte

Nachricht eine Load-Nachricht erwartet. Im Anschluss daran sendet die Karte eine ResAuthenticate-Nachricht an das Ladeterminal. Als Argument bekommt die Nachricht den Wert des Attributs challenge übergeben. Hiermit endet der Protokollschritt der Kopierkarte.

9. Das Ladeterminal empfängt die Nachricht ResAuthenticate und speichert die empfangene Nonce in der neu deklarierten lokalen Variable challenge.
10. Als erstes prüft das Terminal, ob es sich zurzeit im Zustand EXPRESAUTH befindet. Ist dies nicht der Fall, wird die Nachricht nicht akzeptiert und der Protokollschritt endet mit einem FlowFinalNode, der das Auftreten eines Fehlers modelliert. Ist das Terminal im für diesen Schritt richtigen Zustand, wird der Zustand auf EXPRESLOAD gesetzt.
11. Im Anschluss soll der Hashwert aus der Konstanten LOAD, dem Geheimnis, der empfangenen Nonce und dem zu ladenden Betrag gebildet werden. Für die Bildung des Hashwertes muss das Ladeterminal eine lokale Variable vom Typ AuthData (das die Klartextdaten für den Hashwert enthält) deklarieren, ein entsprechendes Objekt von diesem Typ erzeugen und der Variablen zuweisen. Im Anschluss kann dann der Hashwert über dieser Variablen gebildet werden. In einer Action wird die Variable mit Namen tmp vom Typ AuthData deklariert. Dieser Variablen wird ein Objekt vom Typ AuthData zugewiesen, dass mittels des Schlüsselworts create erzeugt wird. Der Ausdruck create AuthData(LOAD,passphrase,challenge,amountToLoad) erzeugt das AuthData-Objekt mit den Argumenten LOAD (die Konstante, die in der Klasse Constants definiert ist und angibt, dass es sich um den Ladevorgang handelt), passphrase (das gemeinsame Geheimnis), challenge (die von der Karte generierte und am Anfang des Protokollschritts empfangene Nonce) sowie dem amountToLoad (der auf die Karte zu ladende Betrag). Die Reihenfolge der Argumente ergibt sich aus der im Klassendiagramm definierten Reihenfolge.
12. In der darauffolgenden Action wird dieses Objekt durch Aufruf der in MEL vordefinierten Methode hash(tmp) gehasht. Das Ergebnis wird der lokalen Variablen hashedauth vom Typ HashedData, die an dieser Stelle deklariert wird, zugewiesen.
13. In der folgenden SendSignalAction sendet das Terminal die Load-Nachricht an die Kopierkarte. Diese enthält den zu ladenden Betrag im Klartext sowie den zuvor erzeugten Hashwert hashedauth. Bei den Argumenten werden erst die Attribute der Klasse angegeben und danach die Assoziation (siehe Abschnitt 4.2.1.7).
14. Die Kopierkarte empfängt die Load-Nachricht und speichert den empfangenen Betrag in der lokalen Variablen value und den Hashwert in der lokalen Variablen hashedauth.
15. Als erstes prüft die Karte, ob sie sich aktuell in dem Zustand EXPLOAD befindet. Ist dies nicht der Fall endet der Protokollschritt mit einem FlowFinalNode.
16. Ist der Zustand korrekt, wird er zurück auf IDLE_CARD gesetzt und im Anschluss überprüft, ob der Hashwert korrekt ist. Hierfür wird im Guard des entsprechenden ControlFlows mittels des Ausdrucks create AuthData(LOAD,passphrase,challenge,value) ein Objekt vom Typ AuthData erzeugt und anschließend der Hashwert gebildet. Die Argumente sind die Konstante LOAD, die Werte der Attribute passphrase und challenge sowie der gerade empfangene und in der Variablen value gespeicherte zu ladende Betrag. Über

diesem Objekt wird dann durch Aufruf der `hash`-Methode der Hashwert gebildet und das Ergebnis mit dem empfangenen Hashwert verglichen.

17. Stimmen beide überein, wird noch geprüft, dass der zu ladende Betrag nicht kleiner oder gleich null ist (da nur positive Beträge auf die Karte geladen werden sollen) und die festgelegte Obergrenze von 20.000 Punkten durch das Aufladen nicht überschritten wird.
18. Ist die Bedingung erfüllt, wird der Kontostand der Karte (Attribut `balance`) um den Betrag `value` erhöht. Das erfolgreiche Aufladen der Karte wird durch Senden einer `ResLoad`-Nachricht zurück an das Ladeterminal bestätigt. Diese enthält den aktuellen Kontostand der Karte.
19. Das Ladeterminal empfängt diese Nachricht und speichert den aktuellen Kontostand der Karte in einer lokalen Variablen `balance`. Es überprüft seinen Zustand. Ist dieser nicht EXPRESLOAD bricht der Protokollschritt ab. Dies ist durch einen `FlowFinalNode` modelliert. Im positiven Fall setzt das Ladeterminal seinen Zustand zurück auf den Initialzustand IDLE_TERMINAL und schickt eine `UResLoad`-Nachricht an den Benutzer `CardOwner`.
20. Diese Nachricht bestätigt dem Benutzer das erfolgreiche Aufladen der Kopierkarte. Die Nachricht enthält den aktuellen Kontostand der Karte (gespeichert in `balance`) sowie den aufgeladenen Geldbetrag (gespeichert im Attribut `amountToLoad`). Der Benutzer empfängt die Nachricht. Damit endet das Protokoll. Dies ist mittels eines `ActivityFinalNodes` modelliert.

4.5.2.3. Bezahlen mit der Kopierkarte

Abbildung 4.29 zeigt das Aktivitätsdiagramm für das Anfertigen von Kopien. An diesem Protokoll sind der Benutzer (`CardOwner`), das Kopiergerät (`CopyingMachine`) sowie die Kopierkarte (`Copycard`) beteiligt.

1. Der Benutzer startet das Protokoll, indem er eine `URequestCopies`-Nachricht an das dem Kopiergerät angeschlossene Terminal schickt. Diese enthält den Betrag, der für die anzufertigenden Kopien von der Karte abzubuchen sind. Damit ist der erste Protokollschritt abgeschlossen.
2. Das Terminal empfängt die `URequestCopies`-Nachricht und speichert den in der Nachricht enthaltenen Betrag in der lokalen Variablen `val`. Dann prüft das Terminal, ob der Wert der lokalen Variablen `val` größer als null ist. Ist diese Bedingung nicht erfüllt, endet der Protokollschritt an dieser Stelle mit einem `FlowFinalNode`. Ist die Überprüfung erfolgreich, wird dem Feld `amountToPay` des Terminals der Wert der lokalen Variablen `val` zugewiesen (`amountToPay := val`). Dies ist notwendig, da dieser Wert noch im nächsten Schritt des Terminals benötigt wird, der Gültigkeitsbereich der lokalen Variablen jedoch am Ende des aktuellen Protokollschritts endet. Beim Bezahlen von Kopien muss sichergestellt sein, dass die Karte echt ist. Wie auch im Protokoll zum Aufladen der Karte wird deshalb ein *Challenge-Response Verfahren* verwendet, um die Karte zu authentifizieren. Das Terminal generiert deshalb eine neue Nonce und speichert diese im Attribut `challenge`. Als nächstes wird der Zustand des Terminals auf EXPRESPAY gesetzt. Der Protokollschritt endet mit dem Senden

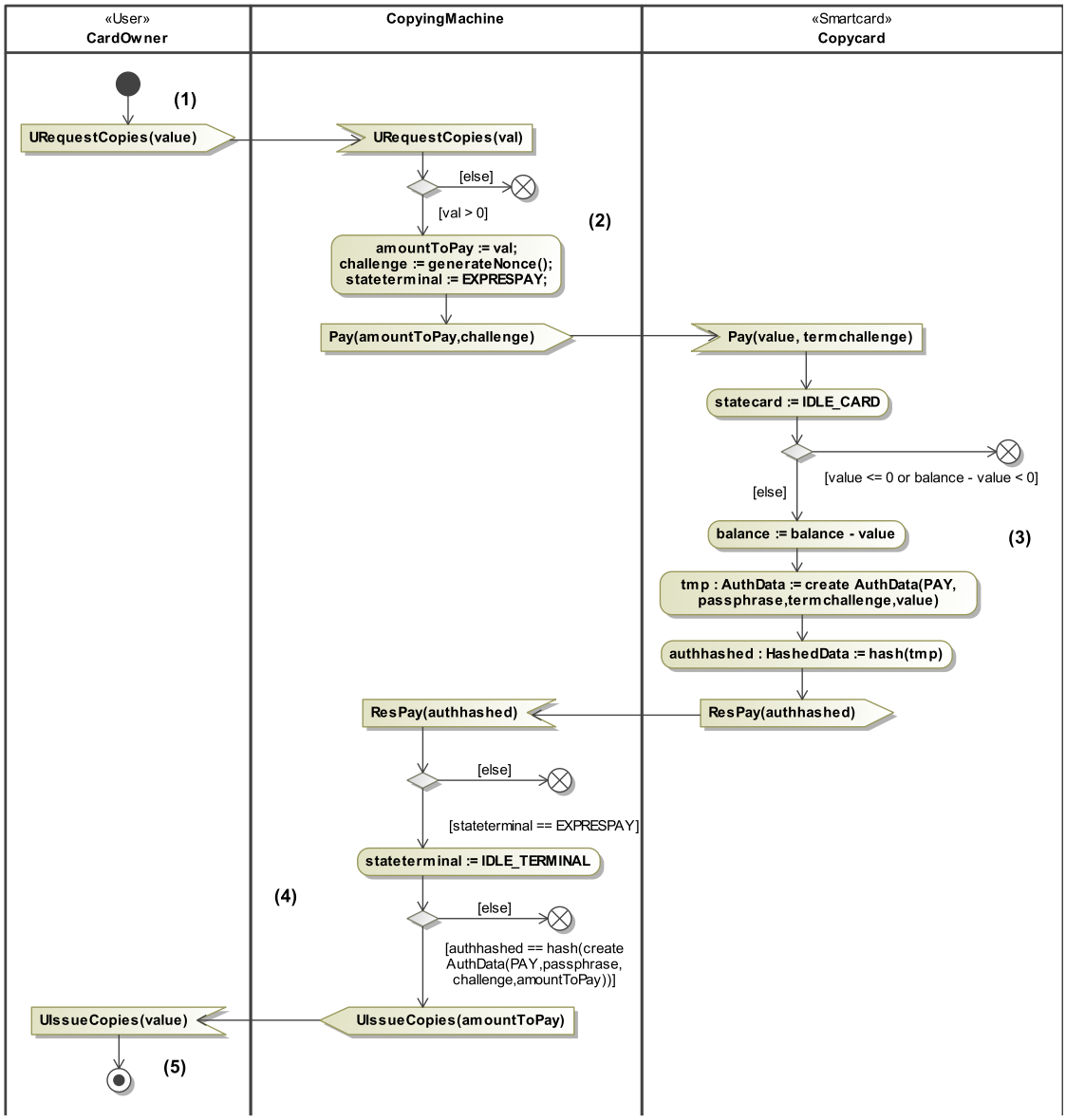

Abbildung 4.29.: Aktivitätsdiagramm für das Bezahlen mit der Kopierkarte

einer `Pay`-Nachricht an die Kopierkarte. Diese Nachricht enthält den Betrag, der abgebucht werden soll (Wert des Attributs `amountToPay`) sowie die zuvor generierte Nonce (gespeichert im Attribut `challenge`).

3. Die Kopierkarte empfängt die `Pay`-Nachricht und speichert die in der Nachricht enthaltenen Werte in den lokalen Variablen `value` und `termchallenge`. Die Kopierkarte setzt ihren Zustand auf IDLE_CARD und überprüft im Anschluss, ob der empfangene Betrag `value` kleiner oder gleich null ist und ob noch genügend Geld auf der Karte ist, um den gewünschten Betrag abzubuchen. Ist dies nicht der Fall, endet das Protokoll mit einem `FlowFinalNode`. Anderenfalls wird der Betrag vom aktuellen Kontostand

subtrahiert. Für eine erfolgreiche Authentifizierung bildet die Kopierkarte nun den Hashwert über der Konstanten PAY, dem Geheimnis `passphrase`, der zuvor empfangenen Nonce `termchallenge` und dem Betrag `value`. Wie auch beim Aufladen der Karte wird zunächst ein Objekt vom Typ `AuthData` erzeugt, das in der lokalen Variable `tmp` gespeichert wird. Über diesem Objekt wird der Hashwert gebildet (`hash(tmp)`) und der neu deklarierten lokalen Variablen `authhashed` vom Typ `HashedData` zugewiesen. Der Hashwert wird dann in einer `ResPay`-Nachricht zurück an das Terminal geschickt und der Protokollschritt der Kopierkarte ist beendet.

4. Das Terminal empfängt die `ResPay`-Nachricht und speichert den darin enthaltenen Hashwert in der lokalen Variablen `authhashed`. Befindet sich das Terminal nicht im Zustand EXPRESPAY, wird der Protokollschritt mit einem `FlowFinalNode` beendet. Ist der Zustand EXPRESPAY wird er zurück auf IDLE_TERMINAL gesetzt und im Anschluss der Hashwert überprüft. Hierfür wird der Hashwert über der Konstanten PAY, der `passphrase`, der `challenge` sowie dem `amountToPay` gebildet und mit dem empfangenen Hashwert `authhashed` verglichen. Stimmen beide überein, weiß das Terminal, dass die Karte echt ist (weil sie das gemeinsame Geheimnis kennt), es sich bei der Nachricht nicht um eine alte Nachricht aus einem vorherigen Protokolllauf handelt (weil die Nonce enthalten ist) und das der richtige Betrag von der Karte abgebucht wurde (weil dieser ebenfalls nochmal im Hashwert enthalten ist). Stimmen die Hashwerte nicht überein, bricht der Protokollschritt ab. Ist der Hashwert korrekt, sendet das Terminal eine `UIssueCopies`-Nachricht an den Kartenbesitzer und schaltet gleichzeitig die Benutzung des Kopiergeräts für die entsprechende Anzahl an Kopien frei. Dies ist nicht explizit modelliert.
5. Die `UIssueCopies`-Nachricht, die den von der Karte abgebuchten Betrag enthält, gibt dem Benutzer eine Bestätigung, dass das Abbuchen erfolgreich war und teilt ihm mit, dass er nun Kopien anfertigen kann. An dieser Stelle endet das Protokoll mit einem `ActivityFinalNode`, der das Ende des Protokolls angibt.

4.6. Verwandte Arbeiten

Die Unified Modeling Language (UML) ist leicht zu erlernen und gilt mittlerweile als de-facto-Standard für die objekt-orientierte Modellierung von Systemen. So ist es nicht überraschend, dass die meisten Ansätze für die Modellierung von Sicherheitsprotokollen die UML verwenden und zum großen Teil um UML-Profile für die Modellierung von Sicherheitsaspekten erweitern. Ausnahmen bilden die Arbeit von Dermott [122], der eine eigene Sprache entwirft (aber nur die Teilnehmer eines Protokolls sowie ihre Interaktion modelliert) und das Werkzeug Autofocus [88], dessen Modellierungssprache in diesem Abschnitt erläutert wird.

In Abschnitt 3.4 wurden bereits einige der Arbeiten, die sich mit der Modellierung von Sicherheitsprotokollen befassen, beschrieben. An dieser Stelle wird nur noch auf die Modellierung dieser Ansätze eingegangen.

UMLSec von Jan Jürjens [20, 89, 99, 100] definiert ein UML-Profil, mit dem kryptographische Protokolle modelliert und Standardsicherheitseigenschaften für die modellierte Anwendung definiert werden können. Der Ansatz unterstützt neben den Sicherheitsprotokollen u.a. auch die Modellierung von Sicherheitsanforderungen, rollenbasierter Zugriffskontrolle und sicherem

Informationsfluss. Jürjens verwendet für die Modellierung Use Case-Diagramme (zur Beschreibung von Sicherheitsanforderungen) und Aktivitätsdiagramme (um detailliertere Use Case-Beschreibungen anzugeben). Für die Modellierung von Sicherheitsprotokollen werden hauptsächlich Klassendiagramme (für die statische Sicht), Deploymentdiagramme (für die physikalische Sicht eines Systems) und Sequenzdiagramme und Zustandsmachinen (für die Beschreibung des dynamischen Verhaltens) verwendet. Die in diesem Kapitel vorgestellte Modellierung einer Anwendung unterscheidet sich von der von Jürjens hauptsächlich bei der Modellierung des dynamischen Teils. Insbesondere die Verwendung von UML-Aktivitätsdiagrammen in Kombination mit der Sprache MEL erlaubt die detaillierte, aber dennoch einfache und implementierungsunabhängige Modellierung der Protokollschritte einer Anwendung. Jürjens verwendet zwar ebenfalls eine zusätzliche Sprache für den Zugriff auf einzelne Teile einer empfangenen Nachricht und die Verwendung kryptographischer Operationen (z.B. zum Verschlüsseln und Signieren), die Protokollschritte selber (z.B. das Abbuchen des Geldbetrags im CEPS-Beispiel [89]) werden jedoch nicht mitmodelliert. Die weiteren Unterschiede zwischen UMLSec und dem in dieser Arbeit beschriebenen Ansatz sind in Abschnitt 3.4 erläutert.

Autofocus [88, 102] ist ein Werkzeug für die Modellierung verteilter Systeme, das bereits 1996 entwickelt wurde. Die Daten der Anwendung werden mittels sogenannter Data Type Definitions (DTDs) definiert. Diese spezifizieren die verwendeten Daten- und Nachrichtentypen. SecureMDD verwendet hierfür UML-Klassendiagramme. Die physikalische Sicht eines Systems wird in Autofocus durch System Structure Diagrams (STDs) modelliert. Diese beschreiben die Komponenten einer Anwendung, Kommunikationskanäle und Ports und ähneln den in SecureMDD verwendeten Deploymentdiagrammen. Das Verhalten der Komponenten einer Anwendung wird in Autofocus durch State Transition Diagrams (STDs) modelliert. Dies sind um lokale Variablen erweiterte endliche Automaten. Ein Zustandsübergang besteht aus Vorbedingungen, Muster für Eingaben, Ausgaben und Änderungen der Werte der lokalen Variablen. Ein STD beschreibt das Verhalten einer Komponente. Zusätzlich zu den STDs beschreiben Extended Event Traces (EETs) beispielhaft die Interaktion zwischen einzelnen Komponenten eines Systems. Die EETs ähneln den UML-Sequenzdiagrammen. Bei Autofocus liegt der Fokus auf der Modellierung des Verhaltens einzelner Komponenten und dessen Verhalten. Für die Modellierung von Smart Card-Anwendungen ist es wichtig, die gesamte Anwendung, d.h. das Zusammenspiel aller Komponenten, zu betrachten. Aus diesem Grund verwendet SecureMDD für die dynamische Modellierung Aktivitätsdiagramme, in denen eine Anwendung aus Funktions- oder Protokollsicht und nicht aus Sicht einzelner Komponenten beschrieben ist.

Bushager et al. stellen in [35] einen Ansatz für die Modellierung einer Smart Card-Anwendung in UML vor. Das UML-Modell kann in ein ausführbares SystemC-Modell übersetzt werden, um darauf Sicherheitstests auszuführen (siehe Abschnitt 7.5). Der Ansatz verwendet Use Case-Diagramme für die Beschreibung der Komponenten und Funktionen der Anwendung und Sequenzdiagramme für die Modellierung der Kommunikation zwischen den einzelnen Komponenten. Es ist jedoch nicht möglich, die entworfenen Protokolle über die Kommunikation hinausgehend, d.h. einschließlich der Verarbeitung der Nachrichten, zu modellieren. Dies ist für die Betrachtung von anwendungsspezifischen Sicherheitseigenschaften jedoch in den allermeisten Fällen notwendig. SecureMDD hingegen ermöglicht die vollständige Modellierung einer Anwendung.

In der Arbeit von Mostowski [140] wird ein Ansatz für die systematische Entwicklung von Java Card-Applets vorgestellt. Für die Modellierung einer Anwendung wird UML, insbesondere UML-Zustandsmaschinen verwendet. Diese beschreiben die möglichen Zustandsänderungen eines Kartenapplets und die Kommandos, die diese Zustandsänderungen bewirken. Im Gegensatz zum SecureMDD-Ansatz wird die Anwendung jedoch nicht vollständig modelliert. Insbesondere die Modellierung der Terminals sowie der kryptographischen Protokolle ist nicht möglich.

Nikseresht et al. [145] modellieren eine Smart Card-Anwendung mit UML. Allerdings wird nur der statische Teil der Anwendung mit Klassendiagrammen modelliert. Die kryptographischen Protokolle werden nicht betrachtet. In dem Papier geht es um die automatische Generierung von Quellcode für die modellierten Teile einer Anwendung. Die Sicherheit der Anwendung wird, anders als in dieser Arbeit, nicht betrachtet.

Kuhlmann und Gogolla erläutern in [112] die Modellierung und Validierung einer Smart Card-Anwendung am Beispiel der Fallstudie Mondex [189], einem elektronischen Bezahlsystem. Die Anwendung wird in UML modelliert. Für den Entwurf der statischen Sicht werden Klassendiagramme verwendet. Mithilfe der Object Constraint Language (OCL) werden Invarianten für die Klassen und Vor- und Nachbedingungen für die definierten UML-Operationen angegeben. Diese Constraints können mit der UML-Based Specification Environment (USE), einem Validierungstool für OCL, validiert werden (siehe Abschnitt 7.5). Die Modellierung der dynamischen Sicht der Anwendung erfolgt nur indirekt über die in den Klassendiagrammen definierten UML-Operationen. Für diese können Constraints wie „Balance ToPurse Increases“ angegeben werden. Mithilfe von Objektdiagrammen können außerdem der Zustand des Systems (bzw. „der Welt“) vor und nach Ausführung der im Klassendiagramm definierten Operationen beschrieben werden.

In [175] evaluieren Smith et al., ob es möglich ist ein Sicherheitsprotokoll mit UML2 ohne Verwendung von Erweiterungen zu modellieren. Sie verwenden für die Modellierung Klassendiagramme und Zustandsmaschinen. Das mit dem RoseRT-Tool erstellte UML-Modell wird mithilfe des Tools in Code übersetzt, der auf Angriffe getestet wird. Bei Verwendung des Ansatzes gibt es jedoch einige Einschränkungen. Das Angreifermodell ist prinzipiell fix und kann nicht, wie in SecureMDD im Deploymentdiagramm, anwendungsspezifisch festgelegt werden. Auch die Anzahl der Komponenteninstanzen ist fix, es gibt immer eine Instanz jeder Komponente (Alice und Bob). Im SecureMDD-Ansatz gibt es im formalen Modell (wie auch in der Realität) immer beliebig viele Smart Cards und Terminals. Der Modellierungsansatz bildet die klassische Schreibweise für kryptographische Protokolle (bei der die Teilnehmer und die ausgetauschten Nachrichten beschrieben werden) auf die Modellierung mit UML ab. Das Verhalten der Anwendung im Fehlerfall und die eigentlichen Protokollschritte, die die Komponenten ausführen, sind nicht beschrieben.

Alle vorgestellten Arbeiten ermöglichen die Modellierung von Sicherheitsprotokollen mit UML. Keine der Arbeiten erlaubt jedoch die vollständige Modellierung einer Anwendung. Der SecureMDD-Ansatz unterstützt diese und ermöglicht dadurch sowohl die Verifikation von anwendungsspezifischen Sicherheitseigenschaften (bei der in der Regel Informationen über die Protokollschritte benötigt werden) als auch die vollautomatische Generierung von lauffähigem Quellcode, der nicht manuell ergänzt werden muss. Dies ist wichtig, da der Code eine Verfeinerung des generierten formalen Modells ist. Wird der Quellcode von Hand vervollständigt, kann diese Verfeinerungsbeziehung nicht mehr garantiert werden.

5 Die Model Extension Language (MEL)

Zusammenfassung: Der SecureMDD-Ansatz ermöglicht die Modellierung einer Anwendung und insbesondere der zugrundeliegenden kryptographischen Protokolle in einem sehr hohen Detailgrad. Die Modellierung mit UML umfasst nicht nur die zwischen den Komponenten ausgetauschten Nachrichten, sondern auch die eigentlichen Protokollschritte, d.h. die Verarbeitung der Nachrichten. Hierfür wird eine domänenspezifische Sprache, genannt Model Extension Language (MEL), definiert, die auf die Modellierung von kryptographischen Protokollen zugeschnitten ist. Diese Sprache ergänzt die in dem Ansatz verwendeten UML-Aktivitätsdiagramme. Dieses Kapitel definiert die abstrakte Syntax sowie die Semantik als auch die konkrete Syntax der Sprache MEL.

Abschnitt 5.1 gibt einen Überblick über die wichtigsten Entwurfsentscheidungen, die beim Design der Sprache MEL getroffen wurden. Abschnitt 5.2 definiert die abstrakte Syntax sowie die Semantik der Sprache und in Abschnitt 5.3 ist die konkrete Syntax beschrieben. In Abschnitt 5.4 sind die in MEL vordefinierten Methoden aufgelistet, Abschnitt 5.5 enthält die für die Modelltransformationen wichtigen Validierungsregeln. Abschließend werden in Abschnitt 5.6 die zu diesem Kapitel verwandten Arbeiten besprochen.

5.1. Entwurfsentscheidungen

In diesem Abschnitt werden die wichtigsten Entscheidungen, die beim Entwurf von MEL getroffen wurden, erläutert und motiviert.

- Beim Entwurf der Sprache stand im Vordergrund, eine einfache Sprache zur Ergänzung von UML-Aktivitätsdiagrammen zu entwickeln. Die Sprache soll für Entwickler, die bereits mit UML gearbeitet haben, vertraut sein und sich intuitiv und ohne großen Einarbeitungsaufwand verwenden lassen. Die Sprache ist eine Erweiterung der UML-Aktivitätsdiagramme. Wann immer möglich, werden jedoch zur Modellierung die Elemente der UML verwendet. So stellt MEL beispielsweise kein if-then-else Statement oder Schleifen zur Verfügung. Stattdessen werden hierfür UML-`DecisionNodes` genutzt.
- Beim Entwurf von MEL wurde darauf geachtet eine möglichst intuitive Sprache zu entwickeln, die es ermöglicht sich auf das Design der Anwendung und nicht so sehr auf

technische Details der Implementierung zu konzentrieren. Beispiele hierfür sind, dass der Zugriff auf statische Felder und Methoden auch ohne Nennung des Klassennamens möglich ist (`state == IDLE` anstatt `state == State.IDLE`) und der Test auf Wertgleichheit von Objekten mittels des ==-Operators möglich ist (anstatt durch Aufrufen der `equals`-Methode wie in Java).

- Die domänenspezifische Sprache MEL ist auf die Entwicklung von sicherheitskritischen Anwendungen zugeschnitten, die auf kryptographischen Protokollen basieren. Der Sprachumfang ist geringer als beispielsweise der von Java. Dennoch bzw. gerade deshalb ist die Sprache gut geeignet für dieses Anwendungsgebiet und einfacher zu erlernen als allgemein verwendbare, umfangreichere Sprachen.
- Die abstrakte Syntax von MEL entspricht im Wesentlichen einer vereinfachten, eingeschränkten Form der abstrakten Syntax der Sprache Java. Einige nicht benötigte Sprachkonstrukte, wie beispielsweise Arrays, werden jedoch nicht unterstützt. Spezielle Ausdrücke, die von MEL unterstützt werden, sind ein `Send`-Ausdruck für das Versenden von Nachrichten sowie ein `Receive`-Ausdruck zum Empfangen einer Nachricht. Diese sind in der abstrakten Syntax von Java nicht enthalten und somit eine echte Erweiterung. MEL ist typsicher, die Typchecks entsprechen denen von Java. MEL wird bei der Codegenerierung in Java- bzw. Java Card-Code übersetzt und ist so präzise wie eine Programmiersprache.
- MEL ist angelehnt an Java, orientiert sich allerdings in der konkreten Syntax an UML. Dadurch wird einem mit UML vertrauten Entwickler die Verwendung der Sprache erleichtert. Beispielsweise verwendet MEL das Schlüsselwort `self` (statt `this` in Java), die binären Operationen `and` und `or` (anstatt && und || in Java) und den Operator := für Zuweisungen (anstatt = in Java). MEL verzichtet auf die Verwendung von Arrays, sondern verwendet in Anlehnung an UML Listen. Um auf eine Liste und darin enthaltene Elemente zuzugreifen, sind in MEL verschiedene Operationen vordefiniert, die hierfür verwendet werden können (siehe 5.4). Der Zugriff auf Arrays ist zwar effizienter als Listen, jedoch ist die Effizienz an dieser Stelle zweitrangig. MEL-Code wird nicht ausgeführt wird, sondern zunächst in Java- bzw. Java Card-Code transformiert. Bei diesem Transformationsschritt werden die in MEL definierten Listen in Arrays übersetzt.
- MEL ist wertbasiert, Referenzen auf Objekte spielen eine untergeordnete Rolle. Grund für diese Entwurfsentscheidung ist die Tatsache, dass Objekte zwischen Komponenten mittels Nachrichten ausgetauscht werden. Um dies technisch umzusetzen, werden die entsprechenden Objekte serialisiert, d.h. in ein Byte Array übersetzt, verschickt und auf Empfängerseite wieder deserialisiert, d.h. das Byte Array wird wieder in ein Objekt umgewandelt. Hierbei werden jedoch nur die Werte der Objekte (de-)serialisiert, d.h. ein Objekt hat auf Empfängerseite eine andere Objektidentität als auf der Seite des Senders. Der Test, ob zwei Objekte dieselbe Identität haben, wird somit für mit SecureMDD modellierte Anwendungen in der Regel fehlschlagen und ist deshalb nicht sinnvoll. Stattdessen basiert MEL auf Wertgleichheit, d.h. zwei Objekte werden als gleich angesehen, wenn sie denselben Typ und alle Attribute und Assoziationsenden die gleichen (Objekt-) Werte haben.
- Eine weitere Besonderheit der Sprache MEL ist ihre Nullfreiheit. Dies bedeutet, dass jedem Attribut und Assoziationsende einer Komponente beim Erzeugen der jeweiligen Komponente ein (Objekt-)Wert zugewiesen wird. Für primitive Datentypen ist dies zu-

nächst ein Defaultwert. Für nichtprimitive Datentypen werden Instanzen erzeugt, deren Attribute und Assoziationen ebenfalls vorinitialisiert werden. Bei der Initialisierung der Komponente werden diese Attribute und Assoziationsenden mit dem entsprechenden Initialwert überschrieben (sofern sie im plattformunabhängigen Modell als „zu initialisieren“ gekennzeichnet sind). Die Nullfreiheit gilt auch für alle lokalen Variablen. Für diese muss bei ihrer Deklaration ein Initialwert angegeben werden. Dies wird beim Validieren des Modells (in Eclipse, vor dem Start der eigentlichen Transformationen) überprüft. Auf diese Weise ist sichergestellt, dass eine Komponente zu jeder Zeit auf eine beliebige Variable zugreifen kann, ohne eine Exception auszulösen.

- MEL stellt eine Reihe von vordefinierten Methoden und Operationen zur Verfügung, die den Entwickler beim Modellieren einer Anwendung möglichst gut unterstützen sollen. Einige dieser Methoden sind rekursiv über den Datenobjekten definiert. Zum Beispiel stellt MEL eine Operation zur Verfügung, die zwei (Objekt-)Werte auf Gleichheit prüft. Diese Operation prüft rekursiv die Gleichheit aller Attribute und Assoziationen der zu vergleichenden Objekte. Um sicherzustellen, dass die in MEL zur Verfügung gestellten Methoden und Operationen immer terminieren, unterstützt die Sprache keine zyklischen Datenstrukturen. Dies hat zur Folge, dass in den Klassendiagrammen keine zyklischen Datenstrukturen definiert werden dürfen (siehe Abschnitt 4.2).
- Bei der Konstruktion der Sprache MEL wurde darauf geachtet, dass bei Ausführung des aus dem UML-Modell generierten (Java- bzw. Java Card-) Codes wann immer möglich keine Java-`Exceptions` auftreten können. Dies ist zum Teil bereits durch die Sprache MEL sichergestellt. Zum Beispiel können aufgrund der Nullfreiheit von MEL bei Ausführung des generierten Codes keine Java `NullPointerExceptions` auftreten. Das Auftreten anderer Arten von Java `Exceptions`, wie z.B. `ArithmeticExceptions` (u.a. bei der Division durch 0), wird dadurch verhindert, dass im generierten Code vor Durchführung der Operation der Test auf 0 durchgeführt wird. Der Grund hierfür ist die für die Korrektheit benötigte Äquivalenz zwischen dem generierten Code und dem formalen Modell. Dies bedeutet u.a. dass sich der Code bei Ausführung bzgl. seines Fehlerverhaltens genauso verhalten muss wie das formale Modell (bzw. die Abstract State Machine) bei Ausführung. Details hierzu finden sich in Kapitel 10.
- Die Sprache MEL ist objektorientiert, verwendet jedoch keine Pointer oder Referenzen. Wird einem Objekt ein anderes Objekt zugewiesen (`a := b`) und anschließend der Wert der rechten Seite (`b`) verändert, hat dies keine Auswirkung auf den Objektwert der Variablen `a`. Dies ist ein Unterschied zu Java und ihrer (impliziten) Pointersemantik. Dort kann es passieren, dass Werte von Variablen (in diesem Fall der Wert von `a`) per Seiteneffekt verändert werden, da nur eine Referenz auf das Originalobjekt `b` gespeichert wird. MEL besitzt eine Kopiersemantik, die bei Zuweisung von einem Objekt eine Kopie erstellt. Daraus ergibt sich, dass in MEL kein Sharing von Daten (d.h. mehrere Objekte speichern Referenzen auf dasselbe Objekt) stattfindet.
- Der in MEL definierte primitive Typ `Number` dient zu Modellierung von ganzen Zahlenwerten, der Wertebereich ist prinzipiell nicht beschränkt. Bei der Codegenerierung wird der Typ `Number`, je nach Zielplattform, in einen `Integer` (für Java-Code) bzw. `short` (für Java Card-Code) übersetzt. Der Wertebereich eines `Integers` reicht von $-2.147.483.648$ bis $2.147.483.647$, der eines `short`-Wertes reicht von -32.768 bis 32.767. Es kann somit passieren, dass es bei Ausführung des generierten Codes aufgrund einer

arithmetischen Operation (wie z.B. Addition, Multiplikation, Pre- und Post-Inkrement) zu einem Over- oder Underflow kommt. Für Integer wird dies bei der Betrachtung von Smart Card-Anwendungen nur in den allerseltensten Fällen auftreten, da der Wertebereich für dieses Anwendungsgebiet in der Regel ausreichend groß ist. Auf Smart Card-Seite kann es dagegen aufgrund des kleinen Wertebereichs des Typs `short` relativ leicht passieren, dass ein ungewollter Over- bzw. Underflow auftritt. Es wurde deshalb die Entscheidung getroffen, bei Auftreten eines Over- bzw. Underflows auf Smart Card-Seite eine Exception zu werfen. Aus diesem Grund führen einige binäre und unäre Operatoren auf Smart Card-Seite (d.h. wenn sie im UML-Modell im Aktivitätsbereich einer Smart Card-Komponente vorkommen) zu einem Programmabbruch durch Werfen einer SecureMDD-Exception, während dies auf Terminalseite nicht der Fall ist. Um auf Smart Card-Seite schon vor Ausführen der Operation festzustellen, ob die Operation eine Exception werfen würde, wurde die Methode `rangeCheck` definiert (siehe Abschnitt 5.4). Dies erlaubt dem Modellierer, die Operation nur durchzuführen, wenn der gültige Wertebereich nicht verlassen wird.

5.2. Abstrakte Syntax und Semantik

Abbildung 5.1 zeigt die abstrakte Syntax von MEL, dargestellt als UML-Modell.

Im Folgenden werden die einzelnen Elemente erläutert:

MELType:
Ein `MELType` ist die abstrakte Oberklasse zur Beschreibung von Typen. Konkrete Klassen sind `ReferenceMELType`, `PrimitiveMELType`, `VoidMELType` und `MELListType`.

ReferenceMELType(id : String):
Ein `ReferenceMELType` ist der Typ für Klassen und Aufzählungen (Enumerations). Gespeichert wird der Bezeichner `id` der Klasse bzw. Enumeration. MEL unterstützt keine Packages, somit reicht ein einzelner String an dieser Stelle aus.

PrimitiveMELType(type : String):
Ein `PrimitiveMELType` ist der Typ für primitive Datentypen. Ein `PrimitiveMELType` speichert den Typ `type` des Datentyps in Form eines Strings. Verwendete primitive Datentypen sind Number, Boolean und String.

VoidMELType:
Ein `VoidMELType` ist der Typ für Ausdrücke, die keinen Rückgabewert haben. Ausdrücke mit Rückgabetyp `VoidMELType` dürfen nur auf Toplevel, d.h. direkt in den UML-Elementen, vorkommen. Als Teilausdrücke innerhalb anderer `Expressions`, z.B. auf der linken oder rechten Seite einer Zuweisung, dürfen diese Ausdrücke nicht verwendet werden.

MELListType(length : Integer, type : MELType):
Ein `MELListType` ist der Typ für Listen. UML-Attribute und Assoziationen mit Multiplizität größer als eins sind vom Typ `MELListType`. In dem Feld `length` vom Typ `Integer`

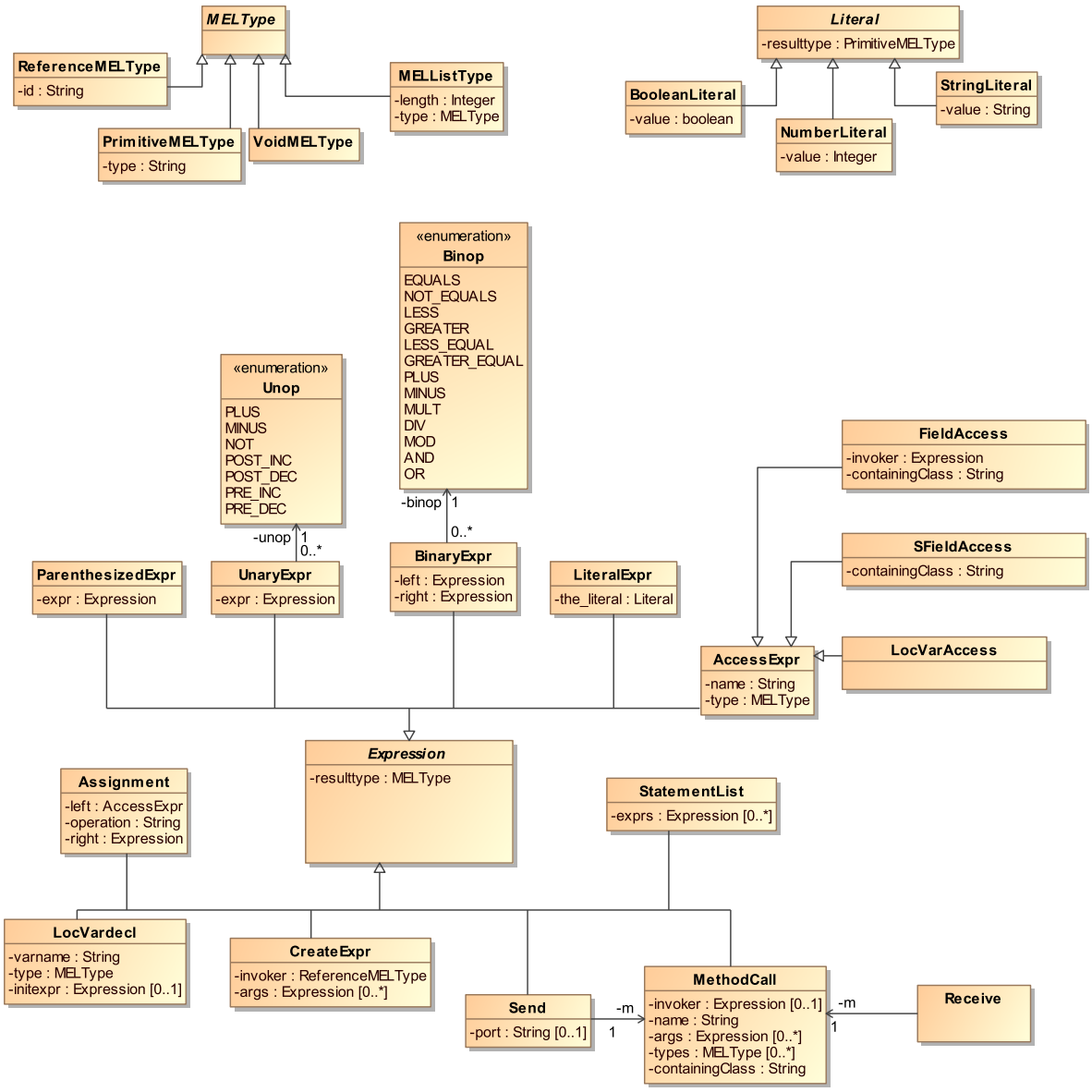

Abbildung 5.1.: Die abstrakte Syntax der Model Extension Language

ist die Länge der Liste gespeichert. Das Feld `type` (vom Typ `MELType`) speichert den Typ, den die Elemente der Liste haben.

Literal(resulttype : PrimitiveMELType):
Ein `Literal` ist die abstrakte Oberklasse für die konkreten Literalklassen `BooleanLiteral`, `StringLiteral` und `NumberLiteral`. Ein Literal beschreibt einen festen, unveränderbaren Wert primitiven Typs. Das Feld `resulttype` vom Typ `PrimitiveMELType` speichert den Rückgabetyp eines `Literals`.

BooleanLiteral(value : boolean):
Ein `BooleanLiteral` repräsentiert ein boolesches Literal. Dieses kann den Wert *true* oder *false* haben. Im Feld `value` wird der Wert des booleschen Literals gespeichert. Der `resulttype` eines `BooleanLiterals` ist vom Typ `PrimitiveMELType`, `type` hat den Wert `boolean`.

StringLiteral(value : String):
Ein `StringLiteral` repräsentiert ein Literal vom Typ `String`. Ein Stringliteral besteht aus Zeichen, die von doppelten Anführungszeichen umschlossen sind. Die gültigen Stringliterale entsprechen den in Java gültigen Stringliteralen. Das Feld `value` speichert den Wert des Literals. Der `resulttype` eines `StringLiterals` ist vom Typ `PrimitiveMELType`, `type` hat den Wert `String`.

NumberLiteral(value : Integer):
Ein `NumberLiteral` repräsentiert ein Literal vom Typ `Number`. Ein Numberliteral ist eine unbeschränkte, ganze Zahl (einschließlich der Zahl Null). Gültige Werte sind diejenigen, die einem gültigen Java Integer entsprechen. Das Feld `value` speichert den Wert des Literals. Der `resulttype` eines `NumberLiterals` ist vom Typ `PrimitiveMELType`, `type` hat den Wert `Number`.

Expression(resulttype : MELType):
Eine `Expression` ist die abstrakte Oberklasse für die Deklaration lokaler Variablen (`Locvarddecl`), eine Zuweisung (`Assignment`), einen geklammerten Ausdruck (`ParenthesizedExpr`), einen unären Ausdruck (`UnaryExpr`), einen binären Ausdruck (`BinaryExpr`), ein Literal (`LiteralExpr`), den Zugriff auf ein Feld oder eine lokale Variable (`AccessExpr`), eine Liste von Statements (`StatementList`), einen Methodenaufruf (`MethodCall`), einen Ausdruck für das Senden einer Nachricht (`Send`) sowie einen Ausdruck zum Erzeugen eines neuen Objekts (`CreateExpr`). Das Feld `resulttype` (vom Typ `MELType`) speichert den Ergebnistyp der Expression.

Locvardecl(varname : String, type : MELType, initexpr : Expression[0..1]):
Eine `Locvardecl` ist die Deklaration einer lokalen Variablen. Diese besteht aus dem Namen der lokalen Variablen, ihrem Typ und optional einer MEL-Expression, die der neu deklarierten Variablen initial zugewiesen wird.
Das Feld `name` vom Typ `String` speichert den Namen der Variablen, das Feld `type` vom Typ `MELType` enthält den Typ der lokalen Variablen und das Feld `initexpr` speichert den Ausdruck, der der lokalen Variablen initial zugewiesen wird. Die Angabe eines initialen Ausdrucks ist optional. Der Rückgabetyp `resulttype` der Deklaration einer lokalen Variablen ist `VoidMELType`.

Die Deklaration einer lokalen Variablen legt eine neue Variable mit dem Namen `name` vom Typ `type` an. Wird der Variablen ein Initialwert zugewiesen, so entspricht die Semantik der einer `StatementList` bestehend aus zwei `Expressions` mit folgenden Eigenschaften: Die erste `Expression` ist die Deklaration der lokalen Variable ohne Initialisierung, die zweite `Expression` ist eine Zuweisung (`Assignment`), in der der Variablen der Ausdruck `initexpr` zugewiesen wird.

Der Sichtbarkeitsbereich einer lokalen Variablen reicht von ihrer Deklaration bis zum Ende des Protokollschrittes. Es gibt drei Möglichkeiten, um einen Protokollschritt zu beenden. Der erste Fall ist das Senden einer Nachricht an eine andere Komponente (und entspricht der Auswertung einer `Send-Expression`). Außerdem endet ein Protokollschritt bei Auftreten eines Fehlers (dies entspricht dem Auftreten eines `FlowFinalNodes` in UML) oder am Protokollende (modelliert durch einen `ActivityFinalNode` in UML). Als Name für eine lokale Variable muss ein neuer Name vergeben werden, d.h. der Name darf noch nicht für eine andere lokale Variable in Benutzung sein. Die Angabe des Typs einer Variablen ist obligatorisch. Als Typ einer Variablen kann jeder im Klassendiagramm definierte oder ein bereits vordefinierter Sicherheitsdatentyp verwendet werden. Der Rückgabetyp des Initialausdrucks muss vom selben Typ wie der des deklarierten Typs sein.

Assignment(left : AccessExpr, right : AccessExpr):
Ein `Assignment` ist eine Zuweisung, bei dem ein Ausdruck (Expression) einem Ausdruck vom Typ `AcessExpr` zugewiesen wird. In MEL gibt es nur den Zuweisungsoperator `:=`, weitere Operatoren wie beispielsweise += aus Java werden nicht unterstützt. Das Feld `left` vom Typ `AccessExpr` ist ein Ausdruck, dem ein Wert zugewiesen wird. Dieser Ausdruck muss der Zugriff auf ein (statisches oder nichtstatisches) Feld oder auf eine lokale Variable sein. Das Feld `right` speichert den Ausdruck, der zugewiesen wird. Der Rückgabetyp eines Assignments ist `VoidMELType`.

Die Semantik eines Assignments ist die Auswertung des rechten Ausdrucks `right` und der Speicherung des Ergebnisses in dem Feld `left`. Dieses ist eine lokale Variable oder ein Feld (d.h. ein Attribut oder gerichtetes Assoziationsende einer UML-Klasse).

Der Rückgabetyp des Ausdrucks `right` muss gleich dem type der `AccessExpr left` sein.

ParenthesizedExpr(expr : Expression):
Eine `ParenthesizedExpr` ist ein geklammerter Ausdruck, bestehend aus einer öffnenden Klammer, einem Ausdruck und einer schließenden Klammer. Das Feld `expr` speichert den Ausdruck, der innerhalb der Klammern steht. Der Rückgabetyp eines geklammerten Ausdrucks ist der Rückgabetyp des Ausdrucks innerhalb der Klammern.

Ein geklammerter Ausdruck hat dieselbe Semantik wie der darin enthaltene Ausdruck `expr`. Der geklammerte Ausdruck wird zu demselben Wert ausgewertet wie der Ausdruck `expr`.

UnaryExpr(unop : Unop, expr : Expression):
Eine `UnaryExpr` ist ein unärer Ausdruck, bestehend aus einem unären Operator und einem Ausdruck. Das Feld `unop` speichert den unären Operator, der auf den Ausdruck `expr` angewendet wird. MEL unterstützt die unären Operatoren NOT, PLUS, MINUS, POST_INC, POST_DEC, PRE_INC und PRE_DEC. Das Feld `expr` vom Typ `Expression` enthält den Ausdruck, auf den der unäre Operator angewandt wird. Der Rückgabetyp eines unären Ausdrucks ist der Rückgabetyp des unären Operators, dieser hat den Rückgabetyp `PrimitiveMELType`.

NOT angewendet auf eine `Expression` liefert das logische Komplement der `Expression` zurück. Wird die `Expression` zu *true* ausgewertet, liefert das logische Komplement *false*. Es

liefert *true*, falls die `Expression` zu *false* ausgwertet wird. Der NOT-Operator kann nur auf Ausdrücke mit Rückgabetyp `PrimitiveMELType` mit `type` Boolean angewendet werden.

Der **PLUS**-Operator angewendet auf eine `Expression` liefert den Rückgabewert dieser zurück. **MINUS** angewendet auf eine `Expression` liefert den negativen Rückgabewert dieser zurück. Für PLUS und MINUS gilt, dass der Rückgabewert der `Expression` vom Typ `PrimitiveMELType` mit `type` Number sein muss. Der Rückgabewert der Operatoren ist ein Wert, keine Variable, auch wenn das Ergebnis der `Expression` eine Variable ist.

Im Folgenden werden die Prä- und Post-Operatoren erläutert. Alle Prä- und Postoperatoren können nur auf Audrücke vom Typ `AccessExpr` angewendet werden. Der Rückgabewert der `AccessExpr` muss eine Variable vom Typ `PrimitiveMELType` mit `type` Number sein. Das Ergebnis der Operation ist ein Wert vom Typ Number, keine Variable.

POST_INC angewendet auf eine `AccessExpr` liefert den unveränderten Wert der `AccessExpr` zurück. Unmittelbar nach Auswertung der `AccessExpr` wird der Wert dieser um eins erhöht und der neue Wert gespeichert. Befindet sich der Ausdruck, der den POST_INC-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Overflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Overflows kommen.

POST_DEC angewendet auf eine `AccessExpr` liefert den unveränderten Wert der `AccessExpr` zurück. Unmittelbar nach Auswertung der `AccessExpr` wird der Wert dieser um eins verringert und der neue Wert gespeichert. Befindet sich der Ausdruck, der den POST_DEC-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Underflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Underflows kommen.

PRE_INC angewendet auf eine `AccessExpr` erhöht den Wert der Variablen, den die `AccessExpr` zurückgibt um Eins, speichert den neuen Wert und gibt diesen zurück. Befindet sich der Ausdruck, der den PRE_INC-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Overflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Overflows kommen.

PRE_DEC angewendet auf eine `AccessExpr` verringert den Wert der Variablen, den die `AccessExpr` zurückgibt um Eins, speichert den neuen Wert und gibt diesen zurück. Befindet sich der Ausdruck, der den PRE_DEC-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Underflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Underflows kommen.

BinaryExpr(left : Expression, binop : Binop, right : Expression):
Eine `BinaryExpr` ist ein binärer Ausdruck, bestehend aus einem linken Ausdruck, einem binären Operator und einem rechten Ausdruck. Das Feld `left` speichert den linken Ausdruck, auf den der Operator angewandt wird. Das Feld `binop` enthält den binären Operator, der an-

gewendet wird. MEL unterstützt die binären Operatoren EQUALS, NOT_EQUALS, LESS, GREATER, LESS_EQUAL, GREATER_EQUAL, PLUS, MINUS, MULT, DIV, REM, AND und OR. Im Feld `right` ist der rechte Ausdruck, auf den der Operator angewandt wird, gespeichert. Der Rückgabetyp eines binären Ausdrucks ist der Rückgabetyp des binären Operators, dieser hat den Rückgabetyp `PrimitiveMELType`.

Der **EQUALS**-Operator angewendet auf zwei `Expressions` `left` und `right` prüft, ob die Rückgabewerte dieser `Expressions` gleich sind. Für den Fall der Gleichheit liefert der Operator *true* zurück, sonst *false*. Geben beide Ausdrücke `left` und `right` nach Auswertung einen Wert vom Typ Number, vom Typ String oder vom Typ Boolean zurück, liefert EQUALS *true*, wenn die Werte gleich sind, sonst wird *false* zurückgegeben. Anders als in Java vergleicht der EQUALS-Operator, angewendet auf zwei Strings, ob diese denselben Wert haben und nicht, ob dasselbe Objekt referenziert wird. Geben beide Ausdrücke nach Auswertung ein Objektwert vom Typ `ReferenceMELType` zurück, liefert EQUALS *true*, wenn beide Objektwerte vom selben Typ sind und alle Attribute und Assoziationsenden gleich sind, d.h. der EQUALS-Operator gibt rekursiv angewendet auf alle Attribute und Assoziationsenden *true* zurück. Sonst wird *false* zurückgegeben. Das heißt, der EQUALS-Operator prüft nicht die Objektidentität, sondern die Wertgleichheit der Objekte. Da die im Klassendiagramm definierten Klassen bzw. die Assoziationen zwischen den Klassen zyklenfrei sind, terminiert die Operation immer.

Der **NOT_EQUALS**-Operator angewendet auf zwei `Expressions` `left` und `right` prüft, ob die Rückgabewerte dieser `Expressions` ungleich sind. Für den Fall der Ungleichheit, liefert der Operator *true* zurück, sonst *false*. Anders als in Java vergleicht der NOT_EQUALS-Operator, angewendet auf zwei Strings, ob diese denselben Wert haben und nicht, ob dasselbe Objekt referenziert wird. Geben beide Ausdrücke `left` und `right` nach Auswertung einen Wert vom Typ Number, vom Typ String oder vom Typ Boolean zurück, liefert NOT_EQUALS *true*, wenn die Werte ungleich sind, sonst wird *false* zurückgegeben. Geben beide Ausdrücke nach Auswertung ein Objektwert vom Typ `ReferenceMELType` zurück, liefert NOT_EQUALS *false*, wenn beide Objektwerte vom selben Typ sind und alle Attribute und Assoziationsenden gleich sind. Sonst wird *true* zurückgegeben. Der NOT_EQUALS-Operator prüft nicht die Objektidentität, sondern die Wertgleichheit der Objekte. Da die im Klassendiagramm definierten Klassen bzw. die Assoziationen zwischen den Klassen zyklenfrei sind, terminiert die Operation immer.

Der Rückgabetyp des EQUALS sowie des NOT_EQUALS Operators ist vom Typ `PrimitiveMELType` mit `type` Boolean. Die Expressions `left` und `right` müssen denselben Rückgabetyp haben. Dieser kann entweder vom Typ `PrimitiveMELType` mit `type` Number, `PrimitiveMELType` mit `type` Boolean, `PrimitiveMELType` mit `type` String oder vom Typ `ReferenceMELType` sein.

LESS angewendet auf zwei Ausdrücke `left` und `right` liefert true zurück, wenn die linke Expression nach Auswertung einen kleineren Wert zurückliefert als der rechte Ausdruck. Sonst wird false zurückgegeben.

GREATER angewendet auf zwei Ausdrücke `left` und `right` liefert true zurück, wenn die linke Expression nach Auswertung einen größeren Wert zurückliefert als der rechte Ausdruck. Sonst wird false zurückgegeben.

LESS_EQUAL angewendet auf zwei Ausdrücke `left` und `right` liefert true zurück, wenn die linke Expression nach Auswertung einen kleineren oder den gleichen Wert zurückliefert wie der rechte Ausdruck. Sonst wird false zurückgegeben.

GREATER_EQUAL angewendet auf zwei Ausdrücke `left` und `right` liefert true zurück, wenn die linke Expression nach Auswertung einen größeren oder den gleichen Wert zurückliefert wie der rechte Ausdruck. Sonst wird false zurückgegeben.

Alle Vergleichsoperatoren LESS, GREATER, LESS_EQUAL und GREATER_EQUAL haben den Rückgabetyp `PrimitiveMELType` mit `type` Boolean. Die Ausdrücke `left` und `right` müssen beide den Rückgabetyp `PrimitiveMELType` mit `type` Number haben.

Der **PLUS**-Operator angewendet auf zwei Ausdrücke mit Rückgabetyp `PrimitiveMELType` mit `type` Number addiert die beiden ausgewerteten Ausdrücke. Das Ergebnis der Operation ist die Summe der beiden ausgewerteten Ausdrücke. Befindet sich der Ausdruck, der den PLUS-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Over- oder Underflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Over- und Underflows kommen.

Der PLUS-Operator angewendet auf zwei Ausdrücke mit Rückgabetyp `PrimitiveMELType` mit `type` String konkateniert zwei Strings. Das Ergebnis der Operation ist ein neuer String, bestehend aus dem ausgewerteten linken Ausdruck konkateniert mit dem ausgewerteten rechten Ausdruck.

Die Ausdrücke `left` und `right` müssen entweder beide den Rückgabetyp `PrimitiveMELType` mit `type` Number haben. In diesem Fall bildet der PLUS-Operator die Summe der beiden Werte. Hat keiner der beiden Ausdrücke den Rückgabetyp `Number`, müssen beide den Rückgabetyp `String` haben. In diesem Fall bildet der PLUS-Operator die Konkatenation über den beiden Strings. Der PLUS-Operator hat den Rückgabetyp `PrimitiveMELType` mit `type` Number (für die Addition zweier Werte) oder mit `type` String (für die Konkatenation zweier Strings).

Der **MINUS**-Operator angewendet auf zwei Ausdrücke bildet die Differenz der beiden Ausdrücke. Das Ergebnis der Operation ist die Subtraktion des zweiten ausgewerteten Ausdrucks vom ersten ausgewerteten Ausdruck, d.h. der erste Ausdruck ist der Minuend, der zweite Ausdruck ist der Subtrahend. Befindet sich der Ausdruck, der den MINUS-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Over- oder Underflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Over- und Underflows kommen.

Beide Ausdrücke müssen den Rückgabetyp `PrimitiveMELType` mit `type` Number besitzen. Der Rückgabetyp der MINUS-Operation ist ebenfalls von diesem Typ.

Der **MULT**-Operator wertet die Ausdrücke `left` und `right` aus und multipliziert sie. Das Ergebnis ist das Produkt beider Ergebniswerte. Befindet sich der Ausdruck, der den MULT-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Over- oder Underflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Over- und Underflows kommen.

Beide Ausdrücke müssen den Rückgabetyp `PrimitiveMELType` mit `type` Number besitzen. Der Rückgabetyp der MULT-Operation ist ebenfalls von diesem Typ.

Der **DIV**-Operator berechnet die ganzzahlige Division zweier Ausdrücke, d.h. er wertet die Ausdrücke `left` und `right` aus und berechnet den Quotienten der Ergebniswerte. Dabei ist der linke Wert der Dividend und der rechte Wert der Divisor. Ist der Divisor Null, wird eine SecureMDD-Exception geworfen und der Protokollschritt beendet. Befindet sich der Ausdruck, der den DIV-Operator enthält, in dem Aktivitätsbereich einer Smart Card-Komponente, wirft die Operation eine SecureMDD-Exception, wenn bei Auswertung ein Over- oder Underflow auftritt. Befindet er sich in dem Aktivitätsbereich einer Terminal- oder Userkomponente, wird keine Exception geworfen. Es kann aber zum Auftreten eines Over- und Underflows kommen.

Beide Ausdrücke müssen den Rückgabetyp `PrimitiveMELType` mit `type` Number besitzen. Der Rückgabetyp der DIV-Operation ist ebenfalls von diesem Typ.

Der **REM**-Operator berechnet den Rest der ganzzahligen Division zweier Ausdrücke, d.h. er wertet die Ausdrücke `left` und `right` aus und berechnet den bei der Division der Ergebniswerte entstehenden Rest. Dabei ist der linke Wert der Dividend und der rechte Wert der Divisor. Ist der Divisor Null, wird eine SecureMDD-Exception geworfen und der Protokollschritt beendet. In allen anderen Fällen verhält sich der Operator so wie der REM-Operator in Java (siehe [63]).

Beide Ausdrücke müssen den Rückgabetyp `PrimitiveMELType` mit `type` Number besitzen. Der Rückgabetyp der REM-Operation ist ebenfalls von diesem Typ.

Der **AND**-Operator berechnet die logische Und-Verknüpfung zweier Ausdrücke `left` und `right`. Der Operator wertet `left` aus und prüft, ob der Ergebniswert *false* ist. Ist dies der Fall, gibt der Operator als Rückgabewert *false* zurück. Liefert der linke Ausdruck nach Auswertung das Ergebnis *true* zurück, wird der rechte Ausdruck ausgewertet. Liefert dieser *true* als Ergebnis, ist das Ergebnis der AND-Operation *true*, sonst *false*.

Der **OR**-Operator berechnet die logische Oder-Verknüpfung zweier Ausdrücke `left` und `right`. Der Operator wertet `left` aus und prüft, ob der Ergebniswert *true* ist. Ist dies der Fall, gibt der Operator als Rückgabewert *true* zurück. Liefert der linke Ausdruck nach Auswertung das Ergebnis *false* zurück, wird der rechte Ausdruck ausgewertet. Liefert dieser *true* als Ergebnis, ist das Ergebnis der OR-Operation *true*, sonst *false*.

Sowohl beim AND- als auch beim OR-Operator müssen beide Ausdrücke den Rückgabetyp `PrimitiveMELType` mit `type` Boolean besitzen. Der Rückgabetyp beider Operationen ist ebenfalls von diesem Typ.

LiteralExpr(the_literal : Literal):
Eine `LiteralExpr` ist ein Ausdruck, der ein Literal repräsentiert. Das Feld `the_literal` speichert das eigentliche Literal. Der `resulttype` einer `LiteralExpr` ist der Rückgabetyp des Literals `the_literal`.

AccessExpr(name : String, type : MELType):
Eine `AccessExpr` beschreibt den Zugriff auf ein (statisches oder nichtstatisches) Feld oder eine lokale Variable. Felder in MEL sind die Attribute und Assoziationsenden einer Klasse

sowie die Attribute einer Enumeration. Eine `AccessExpr` besteht aus dem Namen `name` und dem Typ `type` des Feldes oder der Variablen, auf die zugegriffen wird. Der Rückgabetyp einer `AccessExpr` ist gleich dem Typ `type`.

FieldAccess(invoker : Expression, containingClass : String):
Ein `FieldAccess` ist der Zugriff auf ein (nichtstatisches) Feld. Dieser besteht aus einem aufrufenden Ausdruck `invoker`, der nach Auswertung ein Objekt der Klasse, in der das Feld definiert ist, oder die Instanz einer Subklasse zurückliefert. Die `containingClass` enthält den Namen der Klasse, in der das Feld definiert ist. In der Superklasse `AccessExpr` gespeichert sind außerdem der Name und Typ des Feldes. Der Rückgabetyp des Zugriffs auf ein Feld ist gleich dem Typ des Feldes.

Bei Auswertung eines `FieldAccess` wird zunächst der `invoker` ausgewertet. Dieser liefert ein Objekt der Klasse, in der das Feld mit Namen `name` definiert ist, oder eine Instanz einer Subklasse zurück. Steht der `FieldAccess` auf der linken Seite eines `Assignments` oder als Operand der `UnaryExprs` POST_INC, POST_DEC, PRE_INC und PRE_DEC, wird dieses Feld als Ergebnis zurückgegeben. Anderenfalls wird der Wert des Feldes zurückgegeben.

Die `containingClass` muss ein Feld mit Namen `name` vom Typ `type` definieren. Der `invoker` muss nach Auswertung eine Instanz dieser Klasse oder einer Subklasse zurückgeben.

SFieldAccess(containingClass : String):
Ein `SFieldAccess` ist der Zugriff auf ein statisches Feld. In der `containingClass` ist der Name der Klasse, in der das Feld definiert ist, gespeichert. In der Superklasse `AccessExpr` gespeichert sind außerdem der Name und Typ des Feldes. Der Rückgabetyp des Zugriffs auf ein statisches Feld ist gleich dem Typ des Feldes.

In MEL sind statische Felder nur für die Definition von Konstanten erlaubt. Auf diese wird nur lesend zugegriffen, d.h. anders als beim `FieldAccess`, ist die Verwendung auf der linken Seite einer Zuweisung sowie die Verwendung als Operand für die unären Ausdrücke POST_INC, POST_DEC, PRE_INC und PRE_DEC nicht erlaubt.

Die Auswertung des `SFieldAccess` liefert den Wert des Feldes mit Namen `name` zurück. Die `containingClass` muss ein Feld mit Namen `name` vom Typ `type` definieren.

LocVarAccess():
Ein `LocVarAccess` ist der Zugriff auf eine lokale Variable. Der Name sowie Typ der lokalen Variablen, auf die zugegriffen wird, sind in der Superklasse `AccessExpr` gespeichert. Der Rückgabetyp des Zugriffs auf eine lokale Variable ist gleich dem Typ der lokalen Variablen.

Bei Auswertung eines `LocVarAccess` muss ebenfalls unterschieden werden, ob der Ausdruck auf der linken Seite eines `Assignments` steht bzw. der Operand der `UnaryExprs` POST_INC, POST_DEC, PRE_INC oder PRE_DEC ist oder nicht. Im ersten Fall wird die Variable als Ergebnis zurückgegeben. Anderenfalls wird der Wert der Variablen zurückgegeben.

Die lokale Variable mit Namen `name` vom Typ `type` muss bei Auswertung des `LocVarAccess` sichtbar sein. Der Sichtbarkeitsbereich einer lokalen Variablen beginnt mit ihrer Deklaration (mittels einer `LocVarDecl` oder durch Auswertung eines `Receives`) und endet am Ende des aktuellen Protokollschritts. Es gibt drei Möglichkeiten, um einen Protokollschritt zu beenden. Der erste Fall ist das Senden einer Nachricht an eine andere Komponente

(und entspricht der Auswertung einer Send-Expression). Außerdem endet ein Protokollschritt bei Auftreten eines Fehlers (dies entspricht dem Auftreten eines FlowFinalNodes in UML) oder am Protokollende (modelliert durch einen ActivityFinalNode in UML).

StatementList(exprs : Expression[0..*]):
Eine StatementList ist eine Liste von Ausdrücken. Die StatementList repräsentiert mehrere in einer UML-Action vorkommende Ausdrücke, die durch Semikolon getrennt sind. Im Feld exprs gespeichert sind die Ausdrücke, die in der Liste enthalten sind. Die Liste kann auch leer sein. Der Rückgabetyp einer StatementList ist VoidMELType.

Bei Auswertung einer StatementList werden die in exprs gespeicherten Ausdrücke der Reihe nach ausgewertet.

MethodCall(invoker : Expression[0..1], name : String, args : Expression[0..*], types : MELType[0..*], containingClass : String[0..1]):
Ein MethodCall ist ein Methodenaufruf. Die aufgerufene Methode ist entweder innerhalb einer Klasse im Klassendiagramm definiert, eine vordefinierte Methode, zum Beispiel für kryptographische Operationen oder den Zugriff auf Listen, oder der Aufruf eines Subdiagramms. Das Feld invoker speichert den aufrufenden Ausdruck, der nach Auswertung ein Objekt der Klasse, in der die aufzurufende Methode deklariert ist, oder eine Instanz einer Subklasse zurückgibt. Statische Methoden, Methoden die in den Komponentenklassen definiert sind, sowie Subdiagrammaufrufe haben keinen Invoker. Bei den vordefinierten Operationen haben nur die Methoden, die auf Listen operieren (d.h. Methoden für Assoziationen mit Multiplizität größer als Eins), einen invoker. Der invoker muss eine Klasse sein, die im Klassendiagramm definiert ist. Der invoker darf keine Enumerationklasse und nicht abstrakt sein. Die Klasse invoker darf keine Komponente (≪Smartcard≫-, ≪Terminal≫- oder ≪User≫klasse) sein und muss der aktuellen Komponentenklasse bekannt sein. Die aktuelle Komponentenklasse ist die, in deren Aktivitätsbereich im Aktivitätsdiagramm sich die Action mit der CreateExpr befindet. Eine Klasse ist einer Komponentenklasse bekannt, wenn die Klasse ein Attribut mit Typ dieser Klasse besitzt oder es eine gerichtete Assoziation von der Komponentenklasse zu dieser Klasse gibt. Die Klasse invoker muss eine Datenklasse (hierzu zählen auch die Klassen, die mit Stereotyp ≪PlainData≫, ≪HashData≫ oder ≪SignData≫ annotiert sind) sein.

Das Feld name speichert den Namen der Methode, die aufgerufen wird. Das Feld args enthält die Argumente, mit der die Methode aufgerufen werden soll. Jedes Argument ist eine Expression, die nach Auswertung ein Ergebnis vom Typ des Parameters zurückliefert. Im Feld types gespeichert ist für jede Methode, welche Typen die Parameter besitzen. Das Feld containingClass speichert in welcher Klasse eine Methode deklariert ist. Für Subdiagramme gibt es keine containingClass. Der resulttype eines MethodCalls ist gleich dem Typ des Rückgabewertes der Methode. Gibt die Methode keinen Wert zurück, ist der Rückgabetyp VoidMELType.

Bei Auswertung eines MethodCalls wird zunächst der Methodenaufruf einer Methodensignatur zugeordnet. Hierfür wird der Name name der Methode, der Name der Klasse, in der die Methode deklariert ist (containingClass) sowie die Typen types der formalen Parameter benötigt. Für Subdiagrammaufrufe wird der Methodenname name einem gleichnamigen UML Aktivitätsdiagramm zugeordnet. Anschließend werden die Argumente args des Methodenaufrufs sowie der invoker ausgewertet und die Methode auf der Instanz, die der invoker nach Auswertung als Ergebnis zurückgibt und mit den Argumentwerten ausgeführt.

Die aufzurufende Methode mit Namen `name` muss in der Klasse `containingClass` deklariert sein. Ist die Methode ein Subdiagrammaufruf, muss es ein UML-Aktivitätsdiagramm mit gleichem Namen geben. Die Anzahl der Argumente `args` des `MethodCalls` muss mit der Anzahl der Parameter der Methodendeklaration (bzw. des Subdiagramms) übereinstimmen. Außerdem müssen die Typen der Argumente und der der Parameter übereinstimmen. Ist ein `invoker` vorhanden, muss dieser nach Auswertung eine Instanz der Klasse `containingClass` zurückgeben.

Receive(m : MethodCall):
Ein `Receive` definiert das Empfangen einer Nachricht und kommt innerhalb der UML-Aktivitätsdiagramme in den `AcceptEventAction`-Elementen vor. Diese modellieren das Empfangen einer Nachricht. Syntaktisch gesehen besteht ein `Receive` aus einem `MethodCall m`. Der Methodenaufruf `m` hat keinen `invoker`. Der Name `name` des Methodenaufrufs ist der Name einer Nachrichtenklasse, die im Klassendiagramm definiert ist. Das Feld `args` speichert Namen (Identifiert) für lokale Variablen, die beim Empfangen einer Nachricht deklariert und mit den empfangenen Daten initialisiert werden. Das Feld `types` speichert die Typen der Argumente. Diese entsprechen den Typen der Parameter des entsprechenden Nachrichtenkonstruktors. Die `containginClass` enthält, wie auch das Feld `name`, den Namen der Nachrichtenklasse. Der Rückgabetyp eines `Receive-Ausdrucks` ist `VoidMELType`.

Zu jedem Receive muss es einen passenden `Send`-Ausdruck geben. Informell ist das Receive die Deklaration einer Methode, das passende `Send` ist ein Ausdruck, der diese Methode aufruft. Die Argumente `args` des Receive, gespeichert im MethodCall `m`, sind Identifier für lokale Variablen. Diese werden bei Aufruf der Receive Methode deklariert und mit den Argumenten des Methodenaufrufs initialisiert. Die lokalen Variablen sind bis zum Ende des Protokollschritts sichtbar. Ein Send-Ausdruck „passt“ zu einem Receive, wenn beide wohlgeformt sind und dieselbe `containingClass` haben. Ein Receive kann nicht ausgewertet werden, sondern entspricht einer Methodendeklaration. Diese Methode wird durch einen entsprechenden Send-Ausdruck aufgerufen.

Der `MethodCall m` darf keinen `invoker` haben. Der Name `name` des `MethodCalls` muss gleich der `containingClass` sein und dem Namen einer Nachrichtenklasse, die im Klassendiagramm definiert ist, entsprechen. Die Argumente `args` müssen Identifier (siehe JLS, [63]) sein und von der Anzahl den Parametern des Konstruktors der Nachrichtenklasse entsprechen. Eventuell ist der Konstruktor der Nachrichtenklasse nicht explizit modelliert (siehe Abschnitt 4.2.1).

Send(m : MethodCall, port : String[0..1]):
Ein `Send`-Ausdruck beschreibt das Senden einer Nachricht. Dieser besteht aus einem Methodenaufruf `m`, d.h. dem Aufruf des Konstruktors der Nachrichtenklasse, von der eine Instanz verschickt werden soll sowie der optionalen Angabe eines Ports, über den die Nachricht gesendet werden soll. Die Information, welche Ports es gibt, findet sich im Deploymentdiagramm. Der Rückgabetyp eines `Send-Ausdrucks` ist `VoidMELType`. Ein `Send`-Ausdruck kommt innerhalb der UML-Aktivitätsdiagramme in einer SendSignalAction vor.
Bei Auswertung eines `Send` Ausdrucks wird eine Instanz der Klasse `containingClass` (gespeichert im `MethodCall m`) mit Argumenten `args` erzeugt und anschließend über den Port `port` an eine andere Komponente gesendet, d.h. auf Seiten der anderen Komponente wird

ein entsprechendes, zu dem `Send` passendes, `Receive` aufgerufen.

Der `MethodCall m` darf keinen `invoker` haben. Der Name `name` des `MethodCalls` muss gleich der `containingClass` sein und dem Namen einer Nachrichtenklasse, die im Klassendiagramm definiert ist, entsprechen. Die Argumente `args` müssen von der Anzahl und dem Typ den Parametern des Konstruktors der Nachrichtenklasse entsprechen. Eventuell ist der Konstruktor der Nachrichtenklasse nicht explizit modelliert. Ist der Adressat der Nachricht nicht eindeutig bestimmbar (siehe Kapitel 4.2.2), muss ein Port angegeben sein. Im anderen Fall lässt sich der zu verwendende Port automatisch aus dem Deploymentdiagramm ermitteln und wird dann automatisch verwendet.

CreateExpr(invoker : ReferenceMELType, args : Expression[0..*]):
Eine `CreateExpr` ist ein Ausdruck, der ein Objekt erzeugt. Eine `CreateExpr` kann innerhalb einer UML-Action stehen. Im Feld `invoker` ist die Klasse, von der ein Objekt erzeugt wird, gespeichert. Das Feld `args` speichert die Argumente des Konstruktoraufrufs. Jedes Argument ist eine `Expression`, die nach Auswertung ein Ergebnis vom Typ des Arguments zurückliefert. Der Rückgabetyp einer `CreateExpr` ist gleich dem Typ des `invokers`.

Bei Auswertung einer `CreateExpr` wird eine neue Instanz der Klasse `invoker` erzeugt. Für jedes Feld der Klasse (d.h. jedes Attribut und gerichtetes Assoziationsende) wird ebenfalls eine Instanz vom Typ des entsprechenden Feldes instanziiert. Anschließend werden die Argumente `args` von links nach rechts ausgewertet und der Konstruktor der Klasse auf den Ergebniswerten aufgerufen.

Der `invoker` einer `CreateExpr` muss eine Klasse sein, die im Klassendiagramm definiert ist. Der `invoker` darf keine Enumerationklasse und nicht abstrakt sein. Die Klasse `invoker` darf keine mit ≪Smartcard≫-, ≪Terminal≫- oder ≪User≫ annotierte Klasse sein und muss der aktuellen Komponentenklasse bekannt sein. Die aktuelle Komponentenklasse ist die, in deren Aktivitätsbereich im Aktivitätsdiagramm sich die `Action` mit der `CreateExpr` befindet. Eine Klasse ist einer Komponentenklasse bekannt, wenn die Klasse ein Attribut mit Typ dieser Klasse besitzt oder es eine gerichtete Assoziation von der Komponentenklasse zu dieser Klasse gibt. Die Klasse `invoker` muss eine Datenklasse (hierzu zählen auch die Klassen, die mit Stereotyp ≪PlainData≫, ≪HashData≫ oder ≪SignData≫ annotiert sind) sein.

Primitive Werte (`Number`, `String` und `Boolean`) müssen und dürfen nicht explizit mittels einer `CreateExpr` erzeugt werden. Für die Datentypen `SymmKey`, `PublicKey`, `PrivateKey`, `Nonce` und `Secret` sind eigene Methoden für das Erzeugen entsprechender Instanzen definiert (siehe Abschnitt 5.4). Instanzen der Klassen `EncDataSymm`, `EncDataAsymm`, `MACData`, `SignedData` und `HashedData` können ebenfalls nur durch Aufruf einer vordefinierten Methode zum Verschlüsseln, Bilden des MAC-Wertes, Signieren oder Hashen erzeugt werden (siehe Abschnitt 5.4). Nachrichtenklassen können ebenfalls nicht explizit mittels einer `CreateExpr`, sondern nur mittels einer `Send-Expression` erzeugt werden.

In MEL darf für jede Klasse nur ein Konstruktor definiert sein. Dieser muss die richtige Anzahl an Parametern vom richtigen Typ haben (für jedes Attribut und gerichtetes Assoziationsende der Klasse eines). Die Definition eines Konstruktors für eine UML-Klasse ist in Abschnitt 4.2.1.7 erläutert.

5.3. Konkrete Syntax

In Abbildung 5.2 ist die konkrete Syntax der Sprache MEL beschrieben. Sie ist der konkreten Syntax von Java nachempfunden, an vielen Stellen ist sie jedoch UML-ähnlicher. Ein Beispiel hierfür ist der Verzicht auf Arrays und generische Datentypen (wie sie in Java verwendet werden). Stattdessen werden in MEL abstraktere Datentypen wie Listen verwendet. Weiterhin sind zum Beispiel die Deklaration lokaler Variablen, das Erzeugen von neuen Objekten sowie die konkrete Syntax für Zuweisungen an der Syntax von UML angelehnt. Außerdem verwendet MEL die Schlüsselwörter `self` (this in Java) und `and`, `or`, `not` (anstatt &&, ||, ! in Java).

Die Bindungsstärke der binären Operatoren sowie Assoziativität und Kommutativität der Operatoren sind so definiert wie in Java.

Start = Action | Guard | SendSignalAction | AcceptEventAction

Action = Expr | Stm*
Guard = Expr | **else**
SendSignalAction = Identifier | Identifier **(** ExprList **)** | Identifier **(** ExprList **) via** Identifier
AcceptEventAction = Identifier | Identifier **(** IdentList **)**

Stm = Expr;
ExprList = ε | Expr[, Expr]*
IdentList = ε | Identifier[, Identifier]*
Expr = Locvardecl | Assignment | CreateExpr | MethodCall
| BinaryExpr | UnaryExpr | LiteralExpr | FieldAccess
| Name | **(** Expr **)**

LocvarDecl = Identifier **:** Type | Identifier **:** Type **:=** Expr
Assignment = FieldAccess **:=** Expr | Name **:=** Expr
CreateExpr = **create** Identifier **(** ExprList **)**
MethodCall = Identifier **(** ExprList **)**
| Expr**.**Identifier **(** ExprList **)**
BinaryExpr = Expr Binop Expr
UnaryExpr = Unop Expr | Expr Unop
LiteralExpr = **true** | **false** | NumberLiteral | StringLiteral
FieldAccess = Expr**.**Identifier
Name = Identifier | Name**.**Identifier | **self**

Identifier = *Gültiger Java Identifier (JLS 3.8)*
Type = ReferenceType | **Boolean** | **Number** | **String**
Binop = **==** | **!=** | < | > | <= | >= | **+** | **-** | ***** | **/** | **%**
| **and** | **or**
Unop = **+** | **-** | **not** | **++** | **--**
NumberLiteral = *Gültiges Java Integer Literal (JLS 3.10.1)*
StringLiteral = *Gültiges Java String Literal (JLS 3.10.5)*
ReferenceType = Identifier

Abbildung 5.2.: Die konkrete Syntax der Model Extension Language

Ausdrücke der Sprache MEL können in verschiedenen UML-Aktivitätsdiagrammelementen verwendet werden. Die Benutzung ist innerhalb der Elemente `Action`, `Guard`, `SendSignalNode` sowie `AcceptEventAction` erlaubt, die unterschiedlich behandelt werden.

Action:
Eine `Action` darf entweder eine einzelne `Expression` oder eine Liste von Statements `Stm` beinhalten. In MEL ist ein Statement eine `Expression`, gefolgt von einem Semikolon. Anders als in Java werden Kontrollstrukturen wie if-then-else und Schleifen nicht von MEL unterstützt, diese sollen mit UML-Aktivitätsdiagrammen modelliert werden.

Guard:
Ein `Guard` darf entweder eine `Expression` (die zu einem booleschen Wert ausgewertet wird) oder das Schlüsselwort `else` enthalten. In SecureMDD dürfen nur die ausgehenden Kanten von UML `DecisionNodes` einen MEL `Guard` enthalten, alle anderen UML Guards müssen leer sein (siehe 4.4.2.2).

SendSignalAction:
Eine `SendSignalAction` modelliert das Senden von Nachrichten. Für Nachrichtenklassen ohne Attribute und Assoziationen enthält eine `SendSignalAction` einen `Identifier`. Dieser entspricht dem Namen der im Klassendiagramm definierten Nachrichtenklasse. Hat die Nachrichtenklasse, von der eine Instanz verschickt werden soll, Attribute und/oder Assoziationen, muss die `SendSignalAction` ein `Identifier`, gefolgt von einer öffnenden Klammer **(**, gefolgt von einer Liste von `Expressions` für die Argumente der Nachricht, gefolgt von einer schließenden Klammer **)** sein. Die Argumente müssen ausgewertet denselben Typ haben wie die entsprechenden Attribute oder Assoziationen. Für eine Nachrichtenklasse ohne Attribute und Assoziationen darf die `ExprList` auch leer sein. Ist nicht eindeutig berechenbar wer der Empfänger der Nachricht ist, muss zusätzlich noch ein Port angegeben werden. In diesem Fall muss der SendSignalNode die Form `Identifier` **(** `ExprList` **) via** `Identifier` haben, d.h. die Portangabe steht hinter dem Schlüsselwort `via` und beinhaltet den Namen des Ports. Die Information welche Ports es gibt sowie deren Namen stehen im UML-Deploymentdiagramm (siehe 4.2.2).

AcceptEventAction:
Eine `AcceptEventAction` modelliert das Empfangen einer Nachricht. Für Nachrichten ohne Attribute und Assoziationen enthält die `AcceptEventAction` einen Identifier. Dieser entspricht dem Namen der Nachrichtenklasse und gibt an, dass die aktuelle Komponente (in dessen Aktivitätsbereich die `AcceptEventAction` definiert ist) eine Nachricht von diesem Typ empfängt. Hat eine Nachrichtenklasse Attribute und Assoziationen, muss die `AcceptEventAction` ein `Identifier`, gefolgt von einer öffnenden Klammer **(**, gefolgt von einer Liste von Identifiern, gefolgt von einer schließenden Klammer **)** sein. `Messagename(var1, var2)` wäre eine `AcceptEventAction` für das Empfangen einer Nachricht mit dem Namen *Messagename*. Die `AcceptEventAction` beihnaltet außerdem Namen für zwei lokale Variablen, *var1* und *var2*, die beim Empfangen einer Nachricht deklariert werden. Für eine Nachrichtenklasse ohne Attribute und Assoziationen darf die `IdentList` auch leer sein. Zu einer `AcceptEventAction` muss es einen passenden `SendSignalNode` geben, der eine Nachricht vom gleichen Typ sendet, wie sie in der `AcceptEventAction` empfangen wird. Informell gesehen entspricht die `AcceptEventAction` einer Methodendeklaration, während die `SendSignalAction` den

Aufruf dieser Methode definiert. Beim Empfangen einer Nachricht werden die lokalen Variablen, dessen Namen in der `AcceptEventAction` definiert sind, deklariert. Die Typen der Variablen ergeben sich aus den Typen der entsprechenden Attribute und Assoziationen in der Nachrichtenklasse (die im Klassendiagramm definiert ist). Die Argumente der gesendeten Nachrichteninstanz (die in dem `SendSignalNode` stehen) werden beim Senden einer Nachricht ausgewertet und die Ergebniswerte den lokalen Variablen zugewiesen. Die lokalen Variablen sind bis zum Ende des Protokollschritts sichtbar.

ExprList:
Eine `ExprList` ist eine Liste von Ausdrücken. Diese kann leer sein oder beliebig viele Expressions, jeweils getrennt durch ein Komma, enthalten.

IdentList:
Eine `IdentList` ist eine Liste von `Identifiern`. Diese kann leer sein oder beliebig viele Identifier, jeweils getrennt durch ein Komma, enthalten.

Expr:
Ein Ausdruck `Expr` kann die Deklaration einer lokalen Variablen (`LocvarDecl`), eine Zuweisung (`Assignment`), ein Ausdruck zum Erzeugen eines neuen Objekts (`CreateExpr`), ein Methodenaufruf (`MethodCall`), ein binärer Ausdruck (`BinaryExpr`), ein unärer Ausdruck (`UnaryExpr`), ein Literal (`LiteralExpr`), der Zugriff auf ein Feld (`FieldAccess`), ein Name (`Name`) oder ein geklammerter Ausdruck **(** `Expr` **)** sein.

LocvarDecl:
Eine `LocvarDecl`, d.h. die Deklaration einer lokalen Variablen, muss einen `Identifier`, gefolgt von einem **:**, gefolgt von einem `Type`, enthalten. Optional ist noch die Angabe eines Initialausdrucks möglich. In diesem Fall hat eine `LocvarDecl` die Form `Identifier` **:** `Type` **:=** `Expr`.

Die Variable mit Namen *var* vom Typ *Typ* wird durch den Ausdruck *var : Typ* deklariert. Als Name für eine lokale Variable muss ein neuer Name vergeben werden, d.h. der Name darf noch nicht für eine andere lokale Variable in Benutzung sein. Die Angabe des Typs einer Variablen ist obligatorisch. Als Typ einer Variablen kann jeder im Klassendiagramm definierte oder ein Sicherheitsdatentyp verwendet werden. Um einer Variablen *var* vom Typ *Number* initial den Ausdruck *e* zuzuweisen, schreibt man *var : Number := e*. Der Rückgabetyp des Initialausdrucks muss vom selben Typ wie der deklarierte Typ sein. Die Deklaration einer lokalen Variablen ist in einer UML-`Action` möglich. Lokale Variablen sind bis zum Ende des Protokollschritts, in dem sie definiert werden, sichtbar.

Assignment:
Ein `Assignment` ist die Zuweisung von Ausdrücken an Felder (d.h. Attribute und Assoziationen einer Klasse) sowie an lokale Variablen. Für den Zugriff auf ein (statisches oder nichtstatisches) Feld muss ein `Assignment` die Syntax `FieldAccess` **:=** `Expr` haben. Für den Zugriff auf eine lokale Variable hat ein Assignment die Form `Name` **:=** `Expr`. `FieldAccess` ist der Zugriff auf das Feld, dem ein Wert zugewiesen wird. `Name` ist der Name der lokalen Variablen, der ein Wert zugewiesen wird. Als Zuweisungsoperator ist nur der Operator **:=** erlaubt, Operatoren wie z.B. += in Java sind in MEL nicht möglich. Der Ausdruck `Expr` muss ausgewertet denselben Typ haben wie das Feld bzw. die lokale Variable. Eine Zuweisung ist in einer UML-`Action` möglich.

CreateExpr:
Eine `CreateExpr` muss die Form **create** `Identifier ( ExprList )` haben. Der `Identifier` ist der Name der Klasse, von der ein Objekt erzeugt werden soll. Die `ExprList` ist eine Liste von Ausdrücken, die ausgewertet werden und beim Erzeugen des Objekts dem Konstruktor als Argumente übergeben werden.

Mittels einer `CreateExpr` lassen sich für die in den Klassendiagrammen der Anwendung definierten Datenklassen (hierzu zählen auch die Klassen, die mit Stereotyp ≪PlainData≫, ≪HashData≫ oder ≪SignData≫ annotiert sind) Objekte erzeugen. Das Erzeugen eines Objekts wird in einer `Action` modelliert.
Für Klassen mit max. einer (gerichteten) Assoziation zu einer anderen Datenklasse ist der Konstruktor implizit vorhanden und besteht aus dem Namen der Klasse sowie den Attributen in der Reihenfolge, in der sie definiert sind. Für eine Klasse mit Assoziationen zu mindestens zwei Datenklassen muss der Konstruktor explizit im Klassendiagramm angegeben sein.

Für eine Klasse `C` mit Attributen `att1,...,attn` kann durch Angabe des MEL-Ausdrucks `create C(vatt1,...,vattn)` ein Objekt der Klasse erzeugt werden. `vatt1,...,vattn` sind primitive Werte oder Objektwerte, die den gleichen Typ wie die jeweiligen Attribute haben. Das neu erzeugte Objekt lässt sich an eine Variable oder ein Feld gleichen Typs zuweisen: `var : C := create C(vatt1, ..., vattn)`.

Primitive Werte (vom Typ `Number`, `String` und `Boolean`) müssen nicht mittels `create` erzeugt werden. Für die Datentypen `SymmKey`, `PublicKey`, `PrivateKey`, `Nonce` und `Secret` sind eigene Methoden für das Erzeugen entsprechender Instanzen definiert (siehe Abschnitt 5.4). Objekte, die verschlüsselte, signierte, gehashte Daten oder einen MAC-Wert enthalten, lassen sich ebenfalls nicht explizit mittels des *create*-Schlüsselwortes erzeugen. Neue Instanzen der Klassen `EncDataSymm`, `EncDataAsymm`, `MACData`, `SignedData` und `HashedData` werden nur durch Aufruf einer vordefinierten Methode zum Verschlüsseln, Bilden des MAC-Wertes, Signieren oder Hashen erzeugt. Nachrichtenklassen können ebenfalls nicht durch eine `CreateExpr` erzeugt werden. Das Erzeugen von Nachrichtenobjekten erfolgt in einem `SendSignalNode`.

MethodCall:
Ein Methodenaufruf muss die Syntax `Identifier ( ExprList )` bzw. `Expr . Identifier ( ExprList )` für Methodenaufrufe mit invoker haben. Der `Identifier` ist der Namen der aufzurufenden Methode, die Ausdrücke der `ExprList` werden ausgewertet und bilden die Argumente für den Methodenaufruf. Die der Methode vorangestellte `Expression` ist der invoker, der ausgewertet ein Objekt der Klasse, in der die Methode definiert ist (oder einer Subklasse) zurückgeben muss. Auf diesem Objekt wird die Methode aufgerufen.

In MEL gibt es drei Arten von Methoden. Es gibt manuelle Methoden (die später von Hand implementiert werden müssen), vordefinierte Methoden (für kryptographische Operationen und Listenoperationen, siehe Abschnitt 5.4) sowie Aufrufe von Subdiagrammen, die ebenfalls einem Methodenaufruf entsprechen. Methodenaufrufe sind innerhalb von UML-Actions möglich.

Ist die Methode in einer Datenklasse definiert, ist der Aufruf nur von einem Objekt der Klasse aus möglich. Das bedeutet, dem Methodenaufruf vorangestellt muss ein MEL-Ausdruck stehen, der ausgewertet ein Objekt der Klasse zurückgibt. Ist die Methode `m` mit Parametern `par_a` und `par_b` in der Klasse `A` definiert, ist ein Aufruf der Methode über folgenden MEL-

Ausdruck modellierbar: $expr.m(a, b)$. `expr` ist ein Ausdruck, der nach Auswertung ein Objekt vom Typ `A` zurückliefert, die Argumente `a` und `b` müssen vom selben Typ wie `par_a` und `par_b` sein.

Ist die Methode in einer Klasse `B` mit Stereotyp ≪Manual≫ definiert, ist der Methodenaufruf über den Ausdruck `B.m(a,b)` möglich. Eine manuelle Methode kann einen (Objekt-)Wert zurückliefern. Die Annotation von Klassen mit dem Stereotyp ≪Manual≫ ist nur für statische Klassen erlaubt.

Außerdem stellt MEL eine Reihe von vordefinierten Methoden zur Modellierung von kryptographischen Operationen sowie für den Zugriff auf Listen zur Verfügung. Diese können ebenfalls innerhalb der UML-Actions aufgerufen werden. Eine vordefinierte Methode `m` mit Parametern `par_a` und `par_b` kann durch den Ausdruck `m(a,b)` aufgerufen werden.

Der Aufruf von Subdiagrammen ist in Abschnitt 4.4.2.2 beschrieben.

BinaryExpr:
Eine `BinaryExpr` muss die Form `Expr Binop Expr` haben, d.h. ein binärer Ausdruck besteht aus einem binären Operator sowie zwei MEL-Ausdrücken, auf die der Operator angewendet wird.

UnaryExpr:
Ein unärer Ausdruck besteht aus einem unären Operator `Unop` und sowie einem Ausdruck `Expr` auf den der Operator angewendet wird. Unäre Ausdrücke können Präfixausdrücke sein und die Form `Unop Expr` haben oder Postfixausdrücke sein und von der Form `Expr Unop` sein.

LiteralExpr:
Eine `LiteralExpr` besteht aus den konkreten Symbolen `true` oder `false` (für boolesche Literale) oder einem `NumberLiteral` oder einem `StringLiteral`.

FieldAccess:
Der Zugriff auf ein statisches oder nichtstatisches Feld, d.h. ein Attribut oder Assoziationsende eines Objekts, hat die Form `Expr.Identifier`. Der Zugriff auf das Attribut oder Assoziationsende eines Objekts wird in MEL mit .-Notation angegeben. Die Syntax hierfür ist an die Syntax von Java angelehnt. Der Zugriff auf das Attribut `att` eines Objekts `a` vom Typ `A` wird durch den Ausdruck `a.att` modelliert. Weiterhin ist `expr.att` ein gültiger `FieldAccess`, wenn der Ausdruck `expr` nach Auswertung ein Objekt vom Typ `A` zurückgibt.

Name:
Ein `Name` muss entweder ein `Identifier` sein, die Form `Name.Identifier` haben oder das Schlüsselwort **self** sein.

Das Schlüsselwort **self** kann innerhalb von `Actions` und `Guards` verwendet werden und liefert die gerade aktuelle Komponenteninstanz zurück. Gerade aktuell ist die Komponenteninstanz, in dessen Partition im Aktivitätsdiagramm sich das Schlüsselwort befindet.

Durch die Deklaration und Verwendung lokaler Variablen kann es zu Namenskonflikten zwischen Attributen bzw. Assoziationsenden einer Komponente und lokalen Variablen kommen. Das vordefinierte Schlüsselwort `self` kann verwendet werden, um zwischen dem Zugriff auf die Felder einer Komponente und dem Zugriff auf eine gleichnamige lokale Variable zu unterscheiden. Ist `var` sowohl ein Attribut einer Komponente als auch eine sichtbare lokale Variable, ist durch `self.var` der Zugriff auf das Attribut und durch `var` der Zugriff auf die lokale Variable möglich.

Identifier:
Ein Identifier ist ein gültiger Java Identifier (siehe JLS [63], Kapitel 3.8).

Type:
Ein `Type` definiert die möglichen Typen in MEL. Ein `Type` ist entweder ein `ReferenceType` oder einer der primitiven Typen `Boolean`, `Number` oder `String`.

Binop:
Die konkrete Syntax der binären Operatoren ist wie folgt definiert:

- == : Entspricht dem EQUALS-Operator der abstrakten Syntax.
- != : Entspricht dem NOT_EQUALS-Operator der abstrakten Syntax.
- < : Entspricht dem LESS-Operator der abstrakten Syntax.
- > : Entspricht dem GREATER-Operator der abstrakten Syntax.
- <= : Entspricht dem LESS_EQUAL-Operator der abstrakten Syntax.
- >= : Entspricht dem GREATER_EQUAL-Operator der abstrakten Syntax.
- + : Entspricht dem binären PLUS-Operator der abstrakten Syntax.
- - : Entspricht dem binären MINUS-Operator der abstrakten Syntax.
- * : Entspricht dem MULT-Operator der abstrakten Syntax.
- / : Entspricht dem DIV-Operator der abstrakten Syntax.
- % : Entspricht dem MOD-Operator der abstrakten Syntax.
- and : Entspricht dem AND-Operator der abstrakten Syntax.
- or : Entspricht dem OR-Operator der abstrakten Syntax.

Unop:
Die konkrete Syntax der unären Operatoren ist wie folgt definiert:

- + Präfix-Operator, entspricht dem unären PLUS-Operator der abstrakten Syntax.
- − Präfix-Operator, entspricht dem unären MINUS-Operator der abstrakten Syntax.
- `not` Präfix-Operator, entspricht dem unären MINUS-Operator der abstrakten Syntax.
- ++ : Prä- und Postfixoperator, entspricht den PRÄ_INC und POST_INC-Operatoren der abstrakten Syntax.
- −− : Prä- und Postfixoperator, entspricht den PRÄ_DEC und POST_DEC-Operatoren der abstrakten Syntax.

NumberLiteral:
Ein `NumberLiteral` steht für konkrete Werte vom Typ `Number` und muss einem gültigen Java Integer Literal (siehe JLS, Abschnitt 3.10.1) entsprechen.

StringLiteral:
Ein `StringLiteral` steht für konkrete Werte vom Typ `String` und muss einem gültigen Java String Literal (siehe JLS, Abschnitt 3.10.5) entsprechen.

ReferenceType:
Ein `ReferenceType` ist ein nichtprimitiver Typ, der einer Klasse entspricht. Diese kann entweder im Klassendiagramm definiert sein oder muss in den vordefinierten Datentypen (wie beispielsweise `Nonce`, `Secret`) enthalten sein. Ein `ReferenceType` muss ein `Identifier` sein.

5.4. Standardbibliothek

MEL stellt eine Reihe vordefinierter Datentypen zur Verfügung, die zum Beispiel bei der Deklaration lokaler Variablen als Typangabe verwendet werden können. Diese umfassen die primitiven Datentypen `Number`, `Boolean` und `String` sowie Datentypen, die für die Kryptographie notwendig sind. Die Datentypen entsprechen den Sicherheitsdatentypen, die von SecureMDD auch in UML definiert und in den Klassendiagrammen verwendet werden können (siehe Abschnitt 4.1.2).

Folgende Datentypen sind in MEL definiert:

- `Number`: Zur Repräsentation von (ganzen) Zahlenwerten
- `String`: Zur Repräsentation von Zeichenketten
- `Boolean`: Zur Repräsentation von Wahrheitswerten
- `SymmKey`: Zur Repräsentation von symmetrischen Schlüsseln
- `PublicKey`: Zur Repräsentation von öffentlichen (asymmetrischen) Schlüsseln
- `PrivateKey`: Zur Repräsentation von privaten (asymmetrischen) Schlüsseln
- `KeyPair`: Zur Repräsentation von (asymmetrischen) Schlüsselpaaren
- `Nonce`: Zur Repräsentation von Nonces (einmalig verwendeten Zufallswerten)
- `Secret`: Zur Repräsentation von Werten, die geheim gehalten werden müssen, zum Beispiel Passwörter oder PIN-Nummern.
- `EncDataSymm`: Zur Repräsentation von symmetrisch verschlüsselten Daten
- `EncDataAsymm`: Zur Repräsentation von asymmetrisch verschlüsselten Daten
- `SignedData`: Zur Repräsentation von signierten Daten
- `HashedData`: Zur Repräsentation von Hashwerten
- `MACData`: Zur Repräsentation von MAC-Werten

Weiterhin sind in MEL einige Methoden vordefiniert, die in MEL-Ausdrücken aufgerufen werden können. Um die Modellierung zu vereinfachen, muss für den Aufruf statischer Methoden kein Invoker angegeben werden. Im Folgenden sind die vordefinierten Methoden erläutert:

`static generateNonce() : Nonce`
Die Methode generiert eine neue, bisher noch nicht verwendete zufällige Zahl (Nonce). Der Rückgabewert der Methode ist ein Objekt vom Typ `Nonce`, das die neu generierte Zufallszahl enthält. Die Methode ist statisch und kann ohne Invoker, d.h. durch den Ausdruck `generateNonce()` ohne Nennung eines Klassennamens, aufgerufen werden.

`static generateKey() : SymmKey`
Die Methode generiert einen neuen symmetrischen Schlüssel. Die Methode ist statisch. Der Rückgabewert der Methode ist ein Objekt vom Typ `SymmKey`, der den generierten Schlüssel enthält.

`static generateKeyPair() : KeyPair`
Die Methode generiert ein neues asymmetrisches Schlüsselpaar, bestehend aus einem privaten und einem öffentlichen Schlüssel. Die Methode ist statisch. Der Rückgabewert der Methode ist ein Objekt vom Typ `KeyPair`, das die neu erzeugten Schlüssel enthält. Das Erzeugen von Schlüsseln ist nur auf Terminalseite erlaubt, auf der Smart Card können keine Schlüsselpaare erzeugt werden. Dies ist damit begründet, dass asymmetrische Schlüsselpaare ausschließlich während der Initialisierung erzeugt werden können. Dies geschieht außerhalb der Karte auf Terminalseite. Die Karte bekommt die generierten Schlüssel dann während der Initialisierung mitgeteilt.

`static encrypt(SymmKey sk, PlainData p) : EncDataSymm`
Die Methode implementiet die symmetrische Verschlüsselung des Klartextobjekts `p` mit dem symmetrischen Schlüssel `sk`. Das Objekt `p` ist eine Instanz einer Klasse, die im Klassendiagramm definiert ist und den Stereotyp ≪PlainData≫ trägt. Die Methode ist statisch. Der Rückgabewert der Methode ist ein verschlüsseltes Objekt vom Typ `EncDataSymm`.

`static encrypt(PublicKey pk, PlainData p) : EncDataAsymm`
Die Methode implementiert die asymmetrische Verschlüsselung des Klartextobjekts `p` mit dem öffentlichen Schlüssel `pk`. Das Objekt `p` ist eine Instanz einer Klasse, die im Klassendiagramm definiert ist und den Stereotyp ≪PlainData≫ trägt. Die Methode ist statisch. Der Rückgabewert der Methode ist ein asymmetrisch verschlüsseltes Objekt vom Typ `EncDataAsymm`.

`static decrypt(SymmKey sk, EncDataSymm eds) : PlainData`
Die Methode implementiert die symmetrische Entschlüsselung des verschlüsselten Objekts `eds` vom Typ `EncDataSymm` mit dem symmetrischen Schlüssel `sk`. Das verschlüsselte Objekt muss vorher mit dem gleichen symmetrischen Schlüssel `sk` verschlüsselt worden sein. Die Methode ist statisch. Der Rückgabewert der Methode ist ein Klartextobjekt vom Typ `PlainData`, das zuvor mit dem symmetrischen Schlüssel `sk` verschlüsselt worden ist. Dieses ist eine Instanz einer Klasse, die im Klassendiagramm den Stereotyp `PlainData` trägt.

Wird das verschlüsselte Objekt `eds` mit einem anderen symmetrischen Schlüssel entschlüsselt als zuvor für die Verschlüsselung verwendet wurde, tritt beim Entschlüsseln eine SecureMDD-Exception auf und der Protokolllauf wird abgebrochen. Dies ist äquivalent zum Erreichen eines UML `FlowFinal`-Knotens in dem Aktivitätsbereich, von der aus die Methode aufge-

rufen wurde.

`static decrypt(PrivateKey privk, EncDataAsymm edas) : PlainData`
Die Methode implementiert die asymmetrische Entschlüsselung des verschlüsselten Objekts `edas` vom Typ `EncDataAsymm` mit dem privaten Schlüssel `privk`. Die Methode ist statisch. Der Rückgabewert der Methode ist ein Objekt mit Klartextdaten vom Typ PlainData. Die zugehörige Klasse muss im Klassendiagramm definiert sein und den Stereotyp ≪PlainData≫ tragen.

Das verschlüsselte Objekt muss vorher mit dem zugehörigen öffentlichen Schlüssel verschlüsselt worden sein. Wird das verschlüsselte Objekt `eds` mit einem falschen privaten Schlüssel entschlüsselt, führt dies zum Werfen einer SecureMDD-Exception und damit zum Abbruch des Protokolllaufs.

`static hash(HashData h) : HashedData`
Die Methode implementiert das Bilden des Hashwertes über dem Objekt `h`, das die Daten enthält, über denen der Hashwert gebildet werden soll. Das Objekt ist die Instanz einer Klasse, die den Stereotyp ≪HashData≫ trägt. Die Methode ist statisch. Der Rückgabewert der Methode ist ein Objekt vom Typ `HashedData`, das die gehashten Daten enthält. Das Prüfen zweier Hashwerte (vom Typ `HashedData`) auf Gleichheit ist mit dem binären Operator == möglich.

`static sign(PrivateKey privk, SignData signd) : SignedData`
Die Methode implementiert das Bilden der Signatur über dem Objekt `signd`, das die Instanz einer Klasse mit Stereotyp ≪SignData≫ ist und die zu signierenden Daten enthält. Als Schlüssel wird der `PrivateKey privk` verwendet. Die Methode ist statisch. Die Methode gibt ein Objekt vom Typ `SignedData` zurück, das die signierten Daten enthält.

`static verify(PublicKey pubk,SignedData signature,SignData signd) : Boolean`
Die Methode implementiert das Überprüfen der Signatur des Objekts `signature`. Der Parameter `pubk` (vom Typ `PublicKey`) ist der öffentliche Schlüssel, mit dem die Signatur überprüft wird. Der Parameter `signature` (vom Typ `SignedData`) ist die Signatur, die überprüft werden soll. Der Parameter `signd` enthält die Klartextdaten, über denen die Signatur gebildet wurde. Dieses Objekt ist die Instanz einer Klasse, die mit Stereotyp ≪SignData≫ annotiert ist. Die Methode ist statisch und gibt einen booleschen Wert zurück. Der Rückgabewert ist `true`, wenn die Signatur `signature` durch Signieren des Objekts `signd` mit dem zum Schlüssel `pubk` gehörenden privaten Schlüssel entstanden ist, sonst `false`.

`static computeMAC(Symmkey kenc, Symmkey kmac, PlainData plain) : MACData`
Die Methode implementiert das Bilden des MAC-Wertes des Klartextobjekts `plain` (siehe Abschnitt 4.1.3). Der Parameter `kenc` ist der symmetrische Schlüssel zum Verschlüsseln des Objekts `plain`. Der Parameter `kmac` enthält den symmetrischen Schlüssel zum Verschlüsseln

des Hashwertes. Der Hashwert wird berechnet über dem mit `kenc` verschlüsselten Objekt `plain`. Der Parameter `plain` enthält die Klartextdaten über denen der MAC-Wert gebildet wird. Dieser ist die Instanz einer Klasse, die mit Stereotyp ≪PlainData≫ annotiert ist. Die Methode ist statisch. Der Rückgabewert der Methode ist ein Objekt vom Typ `MACData`. Dieses besteht aus dem mit `kenc` symmetrisch verschlüsselten Objekt `plain` (vom Typ `EncDataSymm`) und dem mit `kmac` verschlüsselten Hashwert des mit `kenc` verschlüsselten Objekts (vom Typ `EncDataSymm`).

`static decryptMAC(SymmKey kenc, SymmKey kmac, MACData macdata) : PlainData`
Überprüfen des MAC-Wertes `macdata` und, bei erfolgreicher Prüfung, Rückgabe des Klartextobjekts (vom Typ `PlainData`). Der Parameter `kenc` enthält den symmetrischen Schlüssel zum Ver- bzw. Entschlüsseln des im Objekt `macdata` enthaltenen Objekts `encrypted` (vom Typ `EncDataSymm`). Der Parameter `kmac` enthält den symmetrischen Schlüssel zum Verschlüsseln des Hashwertes (zur Überprüfung); der Hashwert wird berechnet über dem verschlüsselten Objekt `encrypted`. Der Parameter `macdata` enthält das Objekt, das den MAC-Wert repräsentiert. Die Methode ist statisch. Der Rückgabewert der Methode ist das `PlainData`-Objekt, über dem der MAC-Wert gebildet wurde.

Ist die Überprüfung des MAC-Wertes nicht erfolgreich, führt dies zum Werfen einer SecureMDD-Exception, die den Protokolllauf beendet.

`static rangeCheck(BinaryExpr ex) : Boolean`
Die Methode prüft, ob bei Auswertung des binären Ausdrucks `ex` ein Overflow auftritt. Die Methode ist statisch und gibt `true` zurück, wenn ein Overflow auftritt, ansonsten `false`. Das Ergebnis der Auswertung des binären Ausdrucks muss vom Typ `Number` sein.

`static rangeCheck(UnaryExpr ex) : Boolean`
Die Methode prüft, ob bei Auswertung des unären Ausdrucks `ex` ein Overflow auftritt. Die Methode ist statisch und gibt `true` zurück, wenn ein Overflow auftritt, ansonsten `false`. Das Ergebnis der Auswertung des unären Ausdrucks muss vom Typ `Number` sein.

`generateCertificate(PrivateKey issuerprivkey, Type_Att_1 att_1, Type_Att_2 att_2, ...,Type_Att_n att_n) : C`
Die Methode ist für jede Klasse `C`, die im Klassendiagramm mit Stereotyp ≪Certificate≫ annotiert ist, definiert. Sie ist zwar statisch, muss aber aufgrund der evtl. mehrfach vorkommenden Definition durch Angabe eines Invokers aufgerufen werden. Der Invoker besteht aus dem Namen der Klasse mit Stereotyp ≪Certificate≫. Der erste Parameter ist der private Schlüssel `issuerprivkey` des Zertifikatherausgebers, mit dem das zu erstellende Zertifikat signiert wird. Die weiteren Parameter der Methode sind die Attribute und Assoziationsenden der zu der Zertifikatsklasse gehörenden Datenklasse. Die Reihenfolge der Parameter ist analog zu der Reihenfolge beim Erzeugen einer Instanz einer Klasse definiert. Die Methode gibt ein Instanz der Klasse `C` zurück, dessen Datenobjekt die Argumente att_1, ... att_n enthält und das als Signatur die von dem Zertifikatherausgeber erstellte Signatur (über dem Datenobjekt) speichert. Die Modellierung von Zertifikaten ist in Abschnitt 4.2.1 beschrieben.

`verifyCertificate(PublicKey pubkey) : Boolean`
Die Methode darf auf der Instanz einer Klasse `C` aufgerufen werden, die im Klassendiagramm mit Stereotyp ≪Certificate≫ annotiert ist. Der Invoker des Methodenaufrufs muss also zu einem Objekt der Klasse `C` ausgewertet werden können. Der Parameter `pubkey` gibt den öffentlichen Schlüssel an, mit dem das Zertifikat verifiziert werden kann. Die Methode gibt `true` zurück, wenn das Zertifikat gültig ist, d.h. mit dem zu dem `PublicKey pubkey` gehörenden privaten Schlüssel signiert wurde. Ansonsten wird `false` zurückgegeben.

Vordefinierte Listenoperationen: Ein Attribut oder Assoziationsende einer Komponente mit Multiplizität größer als eins wird in MEL als Liste interpretiert. Für den Zugriff auf diese Felder sind in MEL einige Operationen vordefiniert, die auf dem entsprechenden Feld aufgerufen werden können. Diese sind nachfolgend beschrieben. Der Typ `Element` ist der Typ des Feldes, auf dem die Methode aufgerufen wird. Dieses Feld hat eine Multiplizität größer als Eins.

- `size() : Number`
  Liefert die Anzahl der in der Liste gespeicherten Elemente zurück.
- `hasFree() : Boolean`
  Gibt *true* zurück, wenn die Liste noch leere Einträge hat, sonst *false*.
- `add(Element e) : void`
  Fügt an das Ende der Liste das Element `e` hinzu. Die Operation wirft eine SecureMDD-Exception und beendet den Protokolllauf, wenn die Liste keine freien Elemente mehr hat, d.h. ein Aufruf der Methode `hasFree()` *false* zurückgibt.
- `at(Number index) : Element`
  liefert das Element zurück, das an der Stelle `index` in der Liste gespeichert ist. Das erste Listenelement hat den Index Null. `index` muss kleiner als die Länge der Liste sein (`index < size()`), sonst wird eine SecureMDD-Exception geworfen und der Protokolllauf beendet.
- `contains(Element e) : Boolean`
  Liefert *true* zurück, wenn das Element `e` in der Liste gespeichert ist, sonst *false*.
- `remove(Element e) : void`
  Entfernt das Element `e` aus der Liste (erstes Vorkommen). Wenn das Element `e` nicht in der Liste enthalten ist, tut die Methode nichts.

5.5. Validierungsregeln für die Verwendung von MEL in Aktivitätsdiagrammen

Um die um MEL-Ausdrücke erweiterten Aktivitätsdiagramme validieren und transformieren zu können, sind bei der Modellierung eine Reihe von Einschränkungen zu berücksichtigen. Diese sind im Folgenden aufgezählt.

- MEL ist eine typsichere Sprache. Die Typkorrektheit ist gleich der Typkorrektheit für Java.

- Die Operation `decrypt` zum Entschlüsseln eines Dokuments ist vordefiniert. Diese Methode darf nur auf der rechten Seite einer Zuweisung oder der Deklaration einer lokalen Variablen verwendet werden. Dies ist wichtig, weil das Ergebnis der Verschlüsselungsoperation auf ein Objekt vom Typ der linken Seite gecastet werden muss.
- In einer Statementliste müssen alle MEL-Ausdrücke den Rückgabetyp `VoidMELType` haben.
- Die Deklaration einer lokalen Variablen muss immer auch einen Initialwert haben. Dies ist in der Nullfreiheit der Sprache begründet.
- In einem geschachtelten Ausdruck dürfen keine Ausdrücke mit Rückgabetyp `VoidMELType` vorkommen (z.B. die Deklaration einer lokalen Variablen). Der Grund hierfür sind Probleme bei der Generierung des formalen Modells, insbesondere der Abstract State Machine.
- MEL-Ausdrücke, die eine Exception auslösen können, dürfen nicht in einem geschachtelten Ausdruck vorkommen. Dies ist ebenfalls mit Schwierigkeiten bei der Generierung der formalen Spezifikation begründet.
- Die unären Operatoren PRE_INC, PRE_DEC, POST_INC und POST_DEC dürfen nicht in einem geschachtelten Ausdruck vorkommen, da sie den Wert des Ausdrucks per Seiteneffekt verändern. Dies führt zu Problemen bei der Generierung des formalen Modells, da dieser Seiteneffekt dort auf eine Zuweisung abgebildet werden muss.

5.6. Verwandte Arbeiten

Mir sind keine anderen mit MEL vergleichbaren Sprachen bekannt, die die Modellierung von (Smart Card-)Anwendungen bzw. von kryptographischen Protokollen mit UML unterstützen. Die Besonderheiten von MEL sind, dass die Sprache in UML integriert ist, d.h. in den UML-Elementen direkt verwendet werden kann, und dass das dynamische Verhalten der modellierten Anwendung in einem sehr hohen Detailgrad beschrieben werden kann.

Seit dem Jahr 2011 gibt es den von der OMG [150] vorgeschlagenen und standardisierten *Foundational Subset for Executable UML Models* (abgekürzt *fUML* oder *Foundational UML*) [72]. fUML definiert für eine Teilmenge der UML (Version 2.3) eine wohldefinierte Semantik und ermöglicht so die Ausführung von fUML-Modellen. Aufbauend auf fUML definiert die OMG die textuelle Sprache (ALF) [70]. Dies ist eine Abkürzung für *Action Language for fUML*. ALF erlaubt die Annotation eines fUML-Modells mit textuellen Ausdrücken, um zum Beispiel UML-Operationen näher zu spezifizieren. Die Sprache ist von der Syntax her an der Programmiersprache Java angelehnt. fUML umfasst auch UML-Aktivitätsdiagramme, d.h. dieser Diagrammtyp kann ebenfalls mit ALF-Ausdrücken annotiert werden. Der Unterschied zu der in diesem Kapitel vorgestellten Sprache MEL ist, dass ALF nicht auf sicherheitskritische Anwendungen zugeschnitten ist und keine Unterstützung für die Modellierung von kryptographischen Protokollen, d.h. Operationen zum Entschlüsseln, Signieren etc. bietet.

Teil III.

Generierung von Code und Testfällen

6 Codegenerierung

Zusammenfassung: Aus dem plattformunabhängigen UML-Modell wird automatisch lauffähiger Quellcode für die Smart Cards sowie die Terminals erstellt. Für die Smart Cards wird Java Card-Code [90] generiert. Java Card ist ein Java-Dialekt, der für die Verwendung auf ressourcenbeschränkten Geräten optimiert ist. Für die Terminals wird Java-Code generiert. Technisch ist die Generierung des Codes in zwei Transformationsschritte unterteilt: eine Modell-zu-Modell-Transformation des plattformunabhängigen UML-Modells in die plattformspezifischen Modelle (PSMs) sowie die anschließende Modell-zu-Text-Transformation der PSMs zu Quellcode. Dieses Kapitel erläutert die Generierung des Quellcodes. Der Übersichtlichkeit halber sind beide Transformationsschritte zu einem Schritt zusammengefasst. Der Aufbau der plattformspezifischen Modelle ist im Anhang A beschrieben. Die Beschreibung der Codegenerierung ist in [132] veröffentlicht.

Das Kapitel beginnt in Abschnitt 6.1 mit der Erläuterung der für dieses Kapitel benötigten Grundlagen. Abschnitt 6.2 beschreibt einige allgemeine Herausforderungen und Probleme, die bei der Erarbeitung der Konzepte für die Codegenerierung aufgetreten sind, und stellt die gewählten Lösungen vor. In Abschnitt 6.3 wird die Generierung des Quellcodes für die Terminals sowie die Smart Cards aus dem UML-Modell im Detail vorgestellt und am Beispiel der Kopierkartenanwendung illustriert. Anhang B enthält den kompletten generierten Code dieser Anwendung. Abschnitt 6.4 setzt die in diesem Kapitel erläuterten Arbeiten in den Kontext anderer Forschungsarbeiten, die sich mit der Generierung von Quellcode befassen.

An den Stellen, an denen sich der generierte Java-Code nicht von dem generierten Java Card-Code unterscheidet, wird im Folgenden von Java-Code gesprochen. Gibt es bei der Generierung einen Unterschied, werden die Begriffe Terminal-Code und Smart Card-Code verwendet.

Die implementierten Modelltransformationen sind in dieser Arbeit nicht abgedruckt. Sie können in [129] nachgelesen werden.

6.1. Grundlagen: Java Card

Java Smart Cards beinhalten einen kompletten Rechner mit eigenem Speicher, Prozessor und Kommunikationsmodul. Die Karten besitzen einen manipulationssicheren Speicher, dessen Daten durch die Programmierung der Smart Card geschützt werden. Das heißt, ein Zugriff ist

nur soweit möglich, wie dies vom Programm vorgesehen ist. Dadurch eignen sich Smart Cards ideal für die Verwendung in sicherheitskritischen Anwendungen, bei denen Informationen wie private Schlüssel keinesfalls unkontrolliert in Umlauf geraten dürfen.

Technisch gesehen bestehen Java Smart Cards meist aus einer 8- oder 16-Bit-CPU, die mit 3,7 MHz getaktet wird. An diesen Prozessor angeschlossen ist ein ca. ein Kilobyte großer flüchtiger Arbeitsspeicher sowie 16 Kilobyte oder mehr an nicht-flüchtigem Speicher. Hochleistungs-Smart Cards werden teilweise mit getrennten Krypto-Prozessoren ausgestattet, in seltenen Fällen werden 32-Bit-Prozessoren eingesetzt.

Aufgrund der begrenzten Leistungsfähigkeit der von Java Smart Cards verwendeten Hardware ist der Sprachumfang von Java Card gegenüber dem Sprachumfang der Java Standard Edition [63] erheblich eingeschränkt. Die wesentlichen Einschränkungen bei Java Card [90] sind der Verzicht auf Garbage Collection, mehrdimensionale Arrays, den 32-Bit-Zahlentyp `int`, Gleitkomma-Typen wie `float` und `double`, Strings sowie Multithreading. Daneben enthält die Java Card-API nur einen Bruchteil der Klassen aus der API der Standard Edition, so dass viele in der Standard Edition selbstverständliche Hilfsklassen in Java Card nachimplementiert werden müssen.

Zentrales Element der Implementierung von Java Card-Anwendungen ist das sogenannte Applet. Dieses ist abgeleitet von der abstrakten Klasse `Applet` des `javacard.framework`-Pakets. Die Java Card-Laufzeitumgebung erzeugt eine Instanz des Applets durch Aufruf der `install`-Methode, die zwingend von der Appletklasse implementiert werden muss. Diese Methode beinhaltet den Aufruf der Methode `register()`, wodurch die Anwendung bei der Java Card-Laufzeitzeitumgebung registriert wird.

Bevor das Applet Nachrichten entgegennehmen kann, muss es selektiert werden. Durch die Selektion eines Applets werden alle weiteren Applets auf der Karte deselektiert. Zu einem Zeitpunkt kann also immer nur eine Anwendung Nachrichten empfangen.

Wird eine Nachricht an das selektierte Applet gesendet, so wird die `process`-Methode dieser Applet-Klasse durch die Java Card-Laufzeitumgebung aufgerufen. Als Argument erhält diese Methode eine sogenannte Application Protocol Data Unit (APDU). Dabei handelt es sich im Wesentlichen um ein Byte-Array, das aus einem standardisierten Kopfbereich sowie einem Datenbereich besteht. Im Kopfbereich werden Meta-Daten wie ein Instruktionsbyte (welches das Kommando beschreibt, das an die Karte geschickt wird), zwei Parameter, die Länge des Datenbereichs, und die Anzahl der erwarteten Antwortbytes angegeben. Die Antwort-APDU der Karte besteht aus einem Datenbereich sowie einem zwei Byte langen Statuswort.

6.2. Allgemeines zum generierten Code

Dieser Abschnitt beschreibt einige Besonderheiten und Herausforderungen, die bei der Generierung des Codes für die Smart Cards und zum Teil auch der Terminals berücksichtigt werden mussten. Diese sind im Folgenden kurz aufgelistet und werden im Anschluss erläutert:

1. Der Programmierstil für Java Card-Programme ist in der Regel Byte-Array-basiert und nicht objekt-orientiert. Der im SecureMDD-Ansatz generierte Java Card-Code sollte jedoch dieselbe Struktur und eine möglichst große Ähnlichkeit zu dem UML-Modell der Anwendung haben. Dies hat den Vorteil, dass die Programme besser lesbar und

verständlich sind. Es wurde deshalb eine Möglichkeit gefunden, um objekt-orientierten Java Card-Code auch für die Smart Cards zu generieren und dort auszuführen. Dies ist in Abschnitt 6.2.1 beschrieben.

2. Im Code werden, wie in der Modellierung, die vordefinierten Sicherheitsdatentypen (wie z.B. `SignedData`) verwendet. Diese implementieren die in MEL vordefinierten kryptographischen Operationen (wie z.B. `sign` und `verify`). Das eigentliche Erstellen und Verifizieren der Signatur wird dabei von Methoden, die die Java Crypto-API bzw. die Java Card Crypto-API zur Verfügung stellt, übernommen. Die Abbildung der in UML und MEL definierten Sicherheitsdatentypen und kryptographische Operationen auf entsprechende Klassen und Methoden im Code ist in Abschnitt 6.2.2 erläutert.
3. Ein Protokollschritt initiiert durch das Senden einer Nachricht an eine andere Komponente den nächsten Protokollschritt. Dieser Zusammenhang sowie die Implementierung des Versendens einer Nachricht zwischen Terminal und Smart Card ist in Abschnitt 6.2.3 beschrieben.
4. Die Kommunikation mit einer Smart Card erfolgt in der Implementierung durch Senden einer APDU (d.h. eines Byte-Arrays) an die Karte bzw. an das Terminal. Modelliert ist die Kommunikation durch das Versenden von Nachrichtenobjekten. Die Abbildung der nachrichtenbasierten Kommunikation im UML-Modell auf die Kommunikation über Byte-Arrays ist in Abschnitt 6.2.4 erläutert.
5. Eine Smart Card verfügt nur über relativ wenig Speicher. Außerdem unterstützt Java Card keine Garbage Collection, um Speicherplatz, der von nicht mehr benötigten Objekten belegt wird, wieder freizugeben. Um keine Speicherlecks zu riskieren, sollten Java Card-Programme zur Laufzeit keine Objekte erzeugen. Stattdessen sollte das Anlegen von Objekten, wenn möglich, bereits beim Erzeugen des Applets erfolgen. Der generierte Smart Card-Code verfügt aus diesem Grund über einen Objektmanager, der die in den Protokollschritten benötigten Objekte bereits bei der Erzeugung des Applets erstellt und dann für die Verwendung bereithält. Dieser Mechanismus ist in Abschnitt 6.2.5 beschrieben.

6.2.1. Objekt-orientierte Programmierung von Smart Cards

Auf Chipkarten lauffähiger und von Hand implementierter Java Card-Code ist in der Regel nicht objekt-orientiert und Paradigmen wie Modularisierung und Kapselung von Daten werden ignoriert. Stattdessen basieren Java Card-Programme auf der direkten Manipulation von Byte-Arrays. Dies führt zu sehr unübersichtlichem und schwer lesbarem Quellcode, in dem Fehler nur schwer zu finden sind. Der Java Card-Codeausschnitt in Listing 6.1 soll dies verdeutlichen. Er zeigt einen Ausschnitt aus einem auf Smart Cards basierenden Bezahlsystem, das als Codebeispiel im Java Card Applet Developers Guide [127] vorgestellt wird.

Grund für die Verwendung eines nicht objekt-orientierten Programmierstils ist die Tatsache, dass Smart Cards mit einem Terminal durch Austausch von speziellen Byte-Arrays, die APDUs genannt werden, kommunizieren. Empfängt eine Smart Card eine APDU von einem Terminal, verarbeitet sie diese und sendet als Antwort eine APDU zurück an das Terminal. Da die Nachrichten somit als Byte-Array vorliegen, ist es am einfachsten, innerhalb des Java Card-Programms direkt auf dieses Byte-Array zuzugreifen. Das Programmieren einer

```
1  private void debit(APDU apdu) {
2  byte[] buffer = apdu.getBuffer();
3  if ( ! pin.isValidated() )
4  ISOException.throwIt(ISO.SW_PIN_REQUIRED);
5  byte numBytes = (byte)(buffer[ISO.OFFSET_LC]);
6  byte byteRead = (byte)(apdu.setIncomingAndReceive());
7
8  if (byteRead != 1) ISOException.throwIt(ISO.SW_WRONG_LENGTH);
9  if (( balance - buffer[ISO.OFFSET_CDATA]) < 0 )
10 ISOException.throwIt(SW_NEGATIVE_BALANCE);
11 balance = (byte) (balance - buffer[ISO.OFFSET_CDATA]);
12 }
```

Listing 6.1: Codeausschnitt aus einem klassischen Java Card-Programm

Transformationsschicht, die die Byte-Arrays in entsprechende Nachrichtenobjekte umwandelt und dadurch eine objekt-orientierte Programmierung auf Smart Cards möglich macht, ist von Hand sehr aufwändig zu programmieren und die Programmierung sehr fehleranfällig. Da die Implementierung zudem noch von der konkreten Anwendung bzw. den dort definierten Nachrichtenklassen abhängt, ist es auch nicht möglich, diese für verschiedene Anwendungen wiederzuverwenden.

An dieser Stelle zeigt sich ein großer Vorteil des modellgetriebenen SecureMDD-Ansatzes. Durch die automatische Generierung des Codes ist es möglich die Serialisierung bzw. Deserialisierung der Nachrichtenobjekte und ihrer Unterobjekte in Byte-Arrays automatisch zu generieren (siehe Abschnitt 6.2.4). Infolgedessen ist es ebenfalls möglich, aus dem UML-Modell objekt-orientierten Smart Card-Code zu generieren. Das Ergebnis ist eine direkte Abbildung der in UML modellierten Klassen sowie der mit UML-Aktivitätsdiagrammen modellierten Protokollbeschreibungen in Java Card-Code. Für die Terminals wird ebenfalls objekt-orientierter Java-Code generiert, der direkt aus dem UML-Modell abgeleitet wird.

6.2.2. Implementierung der kryptographischen Datentypen und Operationen

Die in UML verwendeten Sicherheitsdatentypen (siehe Abschnitt 4.1.2) werden bei der Generierung des Codes in entsprechende Java-Klassen übersetzt. Die in MEL vordefinierten kryptographischen Operationen sind als statische Methoden dieser Java-Klassen implementiert. Zum Beispiel implementiert die Klasse `Nonce` die statische Methode `generateNonce()`, die ein neues Objekt vom Typ `Nonce` erzeugt (bzw. im Smart Card-Code vom Objektmanager erhält) und an den Aufrufer zurückgibt. Im Smart Card-Code werden für die Durchführung der eigentlichen kryptographischen Operationen (wie das Generieren einer Nonce oder das Verschlüsseln eines Datums) die entsprechenden Methoden der Java Card Cryptography APIs [90] verwendet. Analog dazu werden im Terminal-Code die Methoden der Java Cryptography Architecture API [91] aufgerufen.

Im Folgenden wird die Implementierung der Klassen `Nonce` (Zufallszahlen), `SymmKey` (symmetrischer Schlüssel) und `SignedData` (Digitale Signatur) sowie der Quellcode für eine Zertifikatklasse erläutert. Die weiteren Sicherheitsdatentypen sind auf ähnliche Weise implementiert. Der Quellcode hierfür ist im Anhang C.1 enthalten. Im Folgenden wird der generierte

Smart Card-Code vorgestellt. Die entsprechende Implementierung auf Terminal-Seite verwendet die Java Cryptography Architecture API und erzeugt die benötigten Objekte direkt über das Java-Schlüsselwort `new` anstatt den Objektmanager zu verwenden. Davon abgesehen sind die Datentypen jedoch äquivalent implementiert.

6.2.2.1. Die Klasse `Nonce`

Listing 6.2 zeigt den generierten Code für die Klasse `Nonce`. Die Implementierung ist unabhängig von der konkreten Anwendung, d.h. sie ist für alle mit SecureMDD entwickelten Anwendungen gleich.

```
1  public class Nonce implements Codeable {
2  public byte[] nonce;
3  private static RandomData rd = RandomData.
4                getInstance(RandomData.ALG_SECURE_RANDOM);
5
6  public Nonce() {
7     nonce = new byte[Store.LENGTHOFNONCE];
8     rd.generateData(nonce,(short)0,(short)nonce.length); }
9
10 public static Nonce generateNonce() {
11    return Store.newNonce().instGenerateNonce(); }
12
13 private Nonce instGenerateNonce() {
14    rd.generateData(nonce,(short)0,(short)nonce.length);
15    return this; }}
```

Listing 6.2: Smart Card-Code der Klasse `Nonce`

Die Klasse implementiert das Interface `Codeable` (siehe Abschnitt 6.2.4). Das Feld `nonce` vom Typ Byte-Array speichert die eigentlichen Zufallsdaten und wird aus dem Attribut `nonce` der UML-Klasse `Nonce` erzeugt. Das statische Feld `rd` wird für das Erzeugen von neuen Zufallswerten benötigt. Die Initialisierung des Feldes mit einer neuen Instanz vom Typ `RandomData` geschieht entweder beim Aufruf des Konstruktors der Klasse oder beim Aufruf der Methode `generateNonce()` (je nachdem, welcher Aufruf zuerst geschieht, Zeilen 6-7 und 12-13). Der Konstruktor der Klasse initialisiert das Feld `nonce` mit einem entsprechend langen Byte-Array (Zeile 8). Die Länge einer Nonce kann über die Konstante `LENGTHOFNONCE` konfiguriert werden. Anschließend wird dieses Byte-Array durch Aufruf der Java Card Crypto API-Methode `generateData` mit zufälligen Daten gefüllt (Zeile 9).

Die Methode `generateNonce()` erhält durch Aufruf der Methode `Store.newNonce()` ein neues `Nonce`-Objekt vom Objektmanager (Zeile 14). Auf diesem ruft sie die Methode `instGenerateNonce()` auf, die das Feld `nonce` des Objekts mit zufällig erzeugten Werten füllt (Zeilen 16-18).

6.2.2.2. Die Klasse `SymmKey`

Listing 6.3 zeigt den Quellcode der Klasse `SymmKey`, die einen symmetrischen Schlüssel repräsentiert. Die Klasse ist ebenfalls unabhängig von der konkreten Anwendung.

```
1  public class SymmKey extends Key implements Codeable {
2  public byte[] key;
3  private DESKey apikey;
4  private static RandomData rd;
5
6  public final static short SYMMKEYLENGTH = 24;
7  private final static byte[] defaultSymmKey = ..;
8
9  public SymmKey() {
10    key = new byte[SYMMKEYLENGTH];
11    Arrays.copy(defaultSymmKey, this.key);
12    apikey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
13          KeyBuilder.LENGTH_DES3_3KEY, false);
14    updateKey();}
15
16 public void updateKey() {
17    apikey.setKey(key,(short)0); }
18
19 public javacard.security.Key toAPIKey() {
20    return apikey; }
21
22 public static SymmKey generateKey() {
23    if (rd == null)
24       rd = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
25    return Store.newSymmKey().instGenerateKey(); }
26
27 private SymmKey instGenerateKey() {
28    rd.generateData(key,(short)0,(short)SymmKey.SYMMKEYLENGTH);
29    updateKey();
30    return this; }}
```

Listing 6.3: Smart Card-Code der Klasse `SymmKey`

Die Klasse `SymmKey` ist, wie auch in UML, von der abstrakten Klasse `Key` abgeleitet. Sie hat ein Feld `key` zur Speicherung der eigentlichen Schlüsseldaten. Die momentane Implementierung der Klasse `SymmKey` verwendet den Verschlüsselungsalgorithmus 3DES [147]. Um in Java Card mithilfe der Cryptography API ein Datum symmetrisch zu verschlüsseln, wird ein Objekt vom Typ `DESKey` verwendet, das den Schlüssel repräsentiert. Dieses Schlüsselobjekt wird in dem Feld `apikey` gespeichert. Wie auch das Generieren von Nonces ist das Generieren eines neuen Schlüssels über die Methode `RandomData.generateData` realisiert. Hierfür besitzt die Klasse ein Feld vom Typ `RandomData`.

Die Konstante `SYMMKEYLENGTH` definiert die Länge eines symmetrischen Schlüssels. Dies sind in der angegebenen Implementierung 24 Bytes (für 3DES). Außerdem definiert die Klasse

SymmKey einen Standardschlüssel (defaultSymmKey), mit dem alle Schlüssel vorinitialisiert werden. Der Konstruktor der Klasse (in Zeile 9-14) erzeugt das Byte-Array für das Feld key und initialisiert es mit dem defaultkey. Anschließend wird das Feld apikey mit einem Objekt vom Typ DESKEY (mit entsprechenden Argumenten für die 3DES Verschlüsselung) initialisiert und die Methode updateKey() aufgerufen.

Die Methode updateKey (Zeilen 16-17) aktualisiert den DESKey mit dem aktuellen Schlüssel key des SymmKey-Objekts. Soll ein Datum mit dem symmetrischen Schlüssel verschlüsselt werden, wird hierfür der DESKey verwendet. Diesen gibt die Methode toAPIKey heraus (Zeilen 19-20).

Das eigentliche Generieren eines neuen Schlüssels findet in der Methode generateKey() statt. Wie auch beim Generieren der Nonce wird zunächst, falls noch nicht existent, eine Instanz der Klasse RandomData erzeugt (Zeilen 23-24). Anschließend wird durch Aufruf der Methode Store.newSymmKey() ein neues Objekt vom Typ SymmKey vom Objektmanager zur Verfügung gestellt (Zeile 25). Auf diesem wird die Methode instGenerateKey() aufgerufen. Sie erzeugt die zufälligen Daten für den symmetrischen Schlüssel und speichert sie in dem Byte-Array key (Zeile 28). Anschließend wird die Methode updateKey() aufgerufen (Zeile 29) und schließlich das neu erzeugte Schlüsselobjekt an den Aufrufer zurückgegeben (Zeile 30).

6.2.2.3. Die Klasse SignedData

Listing 6.4 zeigt die Felder und den Konstruktor der Klasse SignedData, die eine digitale Signatur repräsentiert. Die in SecureMDD verwendeten Signaturen werden mit dem RSA-SHA1-Verfahren [113, 148] erzeugt. Die Implementierung der Klasse ist unabhängig von der konkreten Anwendung.

```
1 public class SignedData implements Codeable {
2 public byte[] signed;
3 private static byte[] tmparray;
4 private static Signature sig;
5 public static final short signedlength = 128;
6
7 public SignedData() {
8    signed = new byte[Store.MAXSIGNEDLENGTH];
9    tmparray = new byte[Store.MAXENCODINGLENGTHSIGNDATA]; }
```

Listing 6.4: Teile des Smart Card-Codes der Klasse SignedData

Die Klasse hat ein Feld signed vom Typ Byte-Array, das nach dem Erstellen der Signatur die Signaturdaten enthält. Während des Konstruktoraufrufs wird dieses Feld mit einem Byte-Array der Länge MAXSIGNEDLENGTH (= 128) belegt. Das in dem Feld tmparray gespeicherte Byte-Array wird ebenfalls im Konstruktor erzeugt. Da die API-Methode zum Signieren die zu signierenden Daten als Byte-Array erwartet, müssen diese zuvor serialisiert werden. Sie werden in tmparray gespeichert. Das Array ist genau so lang, dass die Instanz jeder Klasse mit Stereotyp ≪SignData≫ serialisiert werden kann. Das Feld sig speichert die für das Signieren benötigte Signature-Objekt.

```
1  public static SignedData sign(PrivateKey key, SignData d) {
2  if (sig == null)
3     sig = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1,
4        false);
5  return Store.newSignedData().instSign(key,d);}
6
7  public SignedData instSign(PrivateKey key, SignData d) {
8  Coding.getInstance().encode(d, tmparray);
9  short signlength = Coding.getInstance().getEncodingLength();
10
11 sig.init(key.toAPIKey(), Signature.MODE_SIGN);
12 sig.sign(tmparray,(short)0,signlength,signed,(short)0);
13 return this; }
```

Listing 6.5: Methoden zum Signieren eines Objekts in der Klasse `SignedData`

Die statische Methode `sign` signiert ein Objekt vom Typ `SignData` mit dem privaten Schlüssel `key`. `SignData` ist ein Interface, das aus dem Stereotyp ≪SignData≫ erzeugt wird. Alle Klassen, die im Klassendiagramm mit diesem Stereotyp annotiert sind, implementieren im Code das Interface `SignData`.

Die Methode erzeugt zunächst, falls noch nicht vorhanden, ein `Signature`-Objekt, das von der Java Card Crypto API zum Bilden einer Signatur benötigt wird (Zeilen 2-4). Anschließend wird ein neues Objekt vom Typ `SignedData` vom Objektmanager zur Verfügung gestellt. Auf diesem wird die Methode `instSign` aufgerufen, die die Signatur erstellt und in dem Objekt speichert (Zeile 5).

Diese serialisiert zunächst die zu signierenden Daten `d` (durch Aufruf der Methode `encode` der Klasse `Coding`, siehe Abschnitt 6.2.4) und speichert das Ergebnis in `tmparray` (Zeile 8). Die Variable `signlength` speichert die Länge der serialisierten Daten (Zeile 9). Mithilfe der API-Methoden `init` und `sign` wird die Signatur erstellt und im Feld `signed` gespeichert (Zeilen 11-12).

Die statische Methode `verify` zum Verifizieren einer Signatur ist auf die identische Weise implementiert wie die Methode `sign`. Ihre Implementierung ist in Anhang C.1 angegeben.

6.2.2.4. Implementierung eines Zertifikats

In UML wird eine Zertifikatklasse mit Stereotyp ≪Certificate≫ annotiert und besitzt zwei Assoziationen zu einer Datenklasse, von der eine mit dem Stereotyp ≪signed≫ annotiert ist.

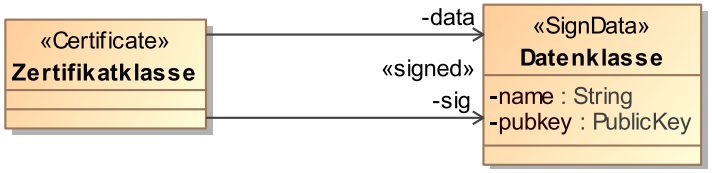

Abbildung 6.1.: Modellierung eines Zertifikats in UML

Die Modellierung eines Zertifikats wurde in Abschnitt 4.2.1.5 beschrieben, Abbildung 6.1 wiederholt diese an einem Beispiel. Bei der Generierung des Codes werden die Zertifikatklasse sowie die Datenklasse in Java-Klassen übersetzt (siehe Listing 6.6).

```
1 public class Zertifikatklasse {
2    public Datenklasse data;
3    SignedData sig; }
4
5 public class Datenklasse implements SignData{
6    byte[] name;
7    PublicKey pubkey; }
```

Listing 6.6: Zertifikatklasse und zugehörige Datenklasse

Die Zertifikatklasse hat zwei Felder: Das erste hat den Namen `data`, trägt den Typ der Datenklasse und speichert die Zertifikatdaten im Klartext. Das zweite Feld mit Namen `sig` ist vom Typ `SignedData` und enthält die Signatur über die Daten.

Die Zertifikatklasse implementiert die statische Methode `generateCertificate()`, die eine Instanz der Klasse, d.h. ein neues Zertifikat, erstellt und als Ergebnis zurückgibt. Die Methode ist abhängig von der Datenklasse, die zu der Zertifikatklasse assoziiert ist. Eine beispielhafte Implementierung ist in Listing 6.7 abgebildet.

```
1 public static Zertifikatklasse generateCertificate(
2    PrivateKey issuerPrivateKey, byte[] name, PublicKey pubkey){
3 Datenklasse dataclass = Store.newDatenklasse(name, pubkey);
4 SignedData s = SignedData.sign(issuerPrivateKey, dataclass);
5 return Store.newZertifikatklasse(s, dataclass);}
```

Listing 6.7: Methode `generateCertificate` der Klasse `Zertifikatklasse`

Die Parameter der Methode sind der private Schlüssel, mit dem die Signatur erstellt wird sowie die Daten, die signiert werden sollen. Dabei wird für jedes Attribut der Datenklasse ein Parameter erzeugt (in diesem Fall `name` und `pubkey`). Die Methode erhält ein neues Objekt vom Typ `Datenklasse` vom Objektmanager, das die übergebenen Daten `name` und `pubkey` enthält (Zeile 3). Dieses Objekt wird anschließend mit dem privaten Schlüssel `privkey` signiert und das Ergebnis in der Variablen `s` vom Typ `SignedData` gespeichert (Zeile 4). Dann wird ein Objekt vom Typ `Zertifikatklasse` erstellt (bzw. vom Objektmanager zur Verfügung gestellt), das die Signatur und das Datenobjekt enthält. Dieses wird an den Aufrufer zurückgegeben (Zeile 5).

Außerdem implementiert die Zertifikatklasse die Methode `verifyCertificate()`, die überprüft, ob ein Zertifikat gültig ist. Listing 6.8 zeigt die Implementierung dieser Methode.

```
1 public boolean verifyCertificate(PublicKey key) {
2       return SignedData.verify(key, sig, data);}
```

Listing 6.8: Methode `verifyCertificate` der Klasse `Zertifikatklasse`

Die Methode bekommt den öffentlichen Schlüssel `key`, gegen den das Zertifikat überprüft werden soll, als Argument übergeben. Sie ruft die Methode `verify` der Klasse `SignedData`

auf, die die Signatur überprüft. Die Argumente dieses Aufrufs sind der übergebene öffentliche Schlüssel `key`, die Signatur, die überprüft werden soll (gespeichert in dem Feld `sig` der Zertifikatklasse) und die Klartextdaten, über denen die Signatur gebildet wurde (gespeichert in dem Feld `data` der Zertifikatklasse).

6.2.3. Kommunikation zwischen einem Terminal und einer Smart Card

Dieser Abschnitt erläutert die Implementierung der Kommunikation zwischen einem Terminal und einer Smart Card. Zunächst wird anhand eines abstrakten Beispiels allgemein die Kommunikation mit einer Smart Card beschrieben. Anschließend werden die generierten Java-Klassen und Methoden, die für die Kommunikation relevant sind, sowie ihre Aufrufreihenfolge erläutert. Eine Besonderheit bei der Kommunikation ist, dass ein Terminal auch mit mehreren Kartenlesern ausgestattet sein kann und somit innerhalb eines Protokolllaufs gleichzeitig mit mehreren Smart Cards kommunizieren kann. Dies ist zum Beispiel in den Fallstudien „Elektronische Gesundheitskarte“ und „Mondex“ der Fall. Hierfür verwendet SecureMDD das Konzept der Ports. Zunächst wird die Kommunikation für den einfachen Fall erläutert, bei der ein Terminal während eines Protokolllaufs mit nur einer Smart Card kommuniziert. Am Ende des Abschnitts wird die Implementierung des Port-Konzepts und die Kommunikation mit mehreren Kartenlesern beschrieben.

Die Kommunikation wird immer von einem Terminal initiiert. Eine Smart Card kann nur auf eine empfangene Nachricht antworten, von sich aus aber keine Nachricht verschicken. Abbildung 6.2 zeigt links ein (vereinfachtes) Aktivitätsdiagramm, das ein Protokoll modelliert. Rechts in der Abbildung ist stark abstrahiert die Realisierung des Versendens von Nachrichten sowie der Zusammenhang zwischen den einzelnen Protokollschritten dargestellt. Die Interaktion mit dem Benutzer ist an dieser Stelle nicht von Bedeutung und deshalb nicht abgebildet.

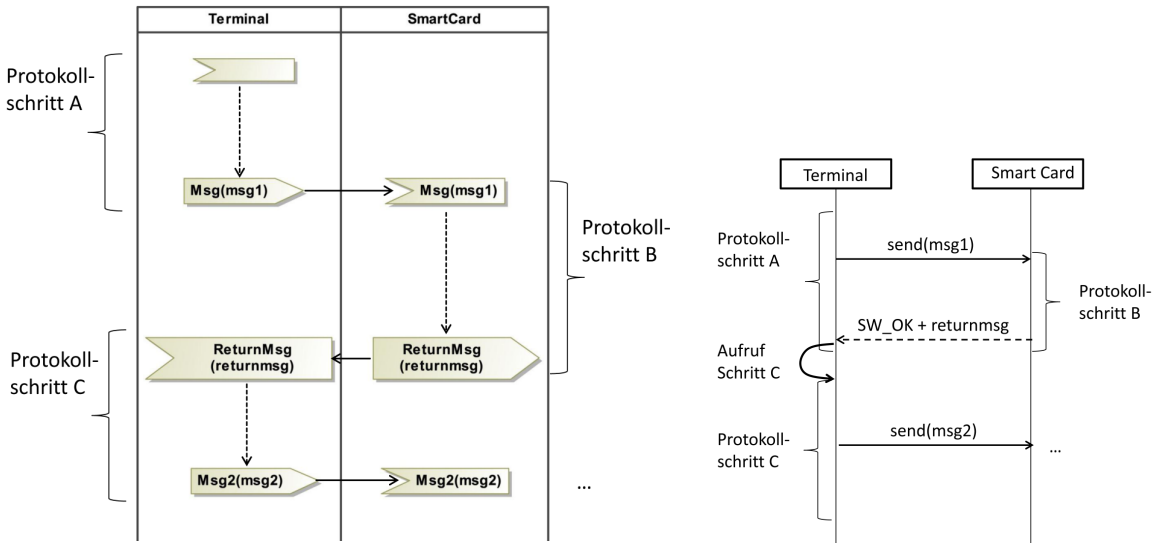

Abbildung 6.2.: Ausschnitt aus einem Aktivitätsdiagramm (links) sowie (abstrakte) Sicht der Implementierung (rechts)

Das im Aktivitätsdiagramm modellierte Protokoll beginnt mit dem Protokollschritt A, der vom Terminal durchgeführt wird. In diesem sendet das Terminal eine Nachricht `msg1` an die Smart Card. Dies geschieht im Code durch Aufruf der Methode `sendMsg`. Die Karte empfängt und verarbeitet die Nachricht in Protokollschritt B. Am Ende des Schrittes sendet sie eine Rückantwort an das Terminal. Die Antwort einer Smart Card besteht immer aus einem Statuswort (einem Short-Wert) und ggf. Antwortdaten in Form eines Byte-Arrays.

Ist das Statuswort „SW_OK“ (0x9000), wurde der Protokollschritt ohne Fehler ausgeführt. Dies bedeutet, dass während des Protokollschritts auf der Karte kein Fehler aufgetreten ist. In diesem Fall entsprechen die Antwortdaten der (serialisierten) Antwortnachricht `returnmsg` der Karte. In der Methode, in der der Protokollschritt A durchlaufen wird, wird dann der Protokollschritt C aufgerufen. Ist auf der Karte während des Protokolllschritts B ein Fehler aufgetreten, sendet die Karte nur einen Fehlercode als Statuswort und keine Antwortdaten zurück.

Die Schritte eines Protokolls sind somit eng miteinander verbunden: Ein Terminal schickt eine Nachricht an die Karte und erhält im fehlerfreien Fall eine Antwortnachricht, die den nächsten Protokollschritt auf Seiten des Terminals beginnt. Tritt auf der Karte während eines Protokollschritts ein Fehler auf, erhält das Terminal einen Fehlercode und der nächste Schritt auf Terminalseite wird nicht mehr ausgeführt. Dadurch wird der gesamte Protokolllauf beendet.

Im Folgenden wird dieser Ablauf detailliert betrachtet. Abbildung 6.3 zeigt die Java-Klassen und -Methoden des generierten Codes, die für die Kommunikation benötigt werden, als UML-Diagramm.

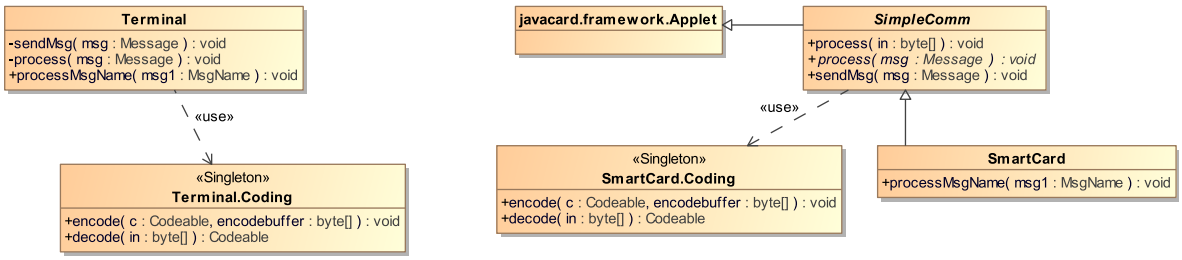

Abbildung 6.3.: Für die Kommunikation relevante Klassen

Die Klasse `Terminal` enthält die Terminalfunktionalität, d.h. die Protokollimplementierung des Terminals. Sie wird aus einer UML-Klasse erzeugt, die im Klassendiagramm mit dem Stereotyp ≪Terminal≫ annotiert ist. Diese Klasse enthält Methoden, um eine Nachricht an eine Smart Card zu senden (`sendMsg`), Nachrichten zu empfangen (`process`) sowie anwendungsabhängige Methoden, die die Protokollschritte durchführen (`processMsgName` für den Nachrichtentyp mit Namen `MsgName`). Die Klasse `Coding` ist für die Serialisierung und Deserialisierung von Nachrichtenobjekten zuständig. Sie ist als Singleton implementiert und wird dafür verwendet, Nachrichtenobjekte vor dem Senden zu serialisieren (Methode `encode`) bzw. ein Byte-Array nach dem Empfang zu deserialiseren (Methode `decode`).

Die Klasse `SmartCard` ist das eigentliche Java Card-Applet. Sie ist von der Klasse `SimpleComm` abgeleitet, die wiederum eine Subklasse der Klasse

`javacard.framework.Applet` ist. Die Klasse `SmartCard` wird aus einer UML-Klasse generiert, die mit Stereotyp ≪Smartcard≫ annotiert ist. Die Klasse `SimpleComm` kapselt die Kommunikation mit der Karte und wird bei der Generierung des Codes zusätzlich zu den schon in UML modellierten Klassen generiert. Sie definiert die Methode `process(in:byte[])`,die in der Klasse `Applet` als abstrakte Methode deklariert ist. Sie wird von der Java Card-Laufzeitumgebung aufgerufen, wenn eine APDU an das Applet gesendet wird. Für das Verarbeiten einer Nachricht gibt es außerdem die Methode `process(msg:Message)`, für das Senden einer Nachricht die Methode `sendMsg(msg:Message)`. Wie auch auf Terminalseite implementiert die Klasse `SmartCard` zusätzlich Methoden für jeden Nachrichtentyp, die den Code für die einzelnen Protokollschritte beinhalten (`processMsgName`). Analog zu der Terminalseite gibt es auch auf der Smart Card die Klasse `Coding`, die für die (De-)Serialisierung der Nachrichtenobjekte zuständig ist.

Die Aufrufreihenfolge dieser Methoden ist in dem Sequenzdiagramm in Abb. 6.4 dargestellt und wird im Folgenden kurz beschrieben. Die Kommunikation ist zu einem großen Teil unabhängig von der konkreten Anwendung. Lediglich das eigentliche Verarbeiten einer empfangenen Nachricht (d.h. die eigentlichen Protokollschritte) ist anwendungsspezifisch und deshalb für jede Anwendung in einem UML-Aktivitätsdiagramm modelliert.

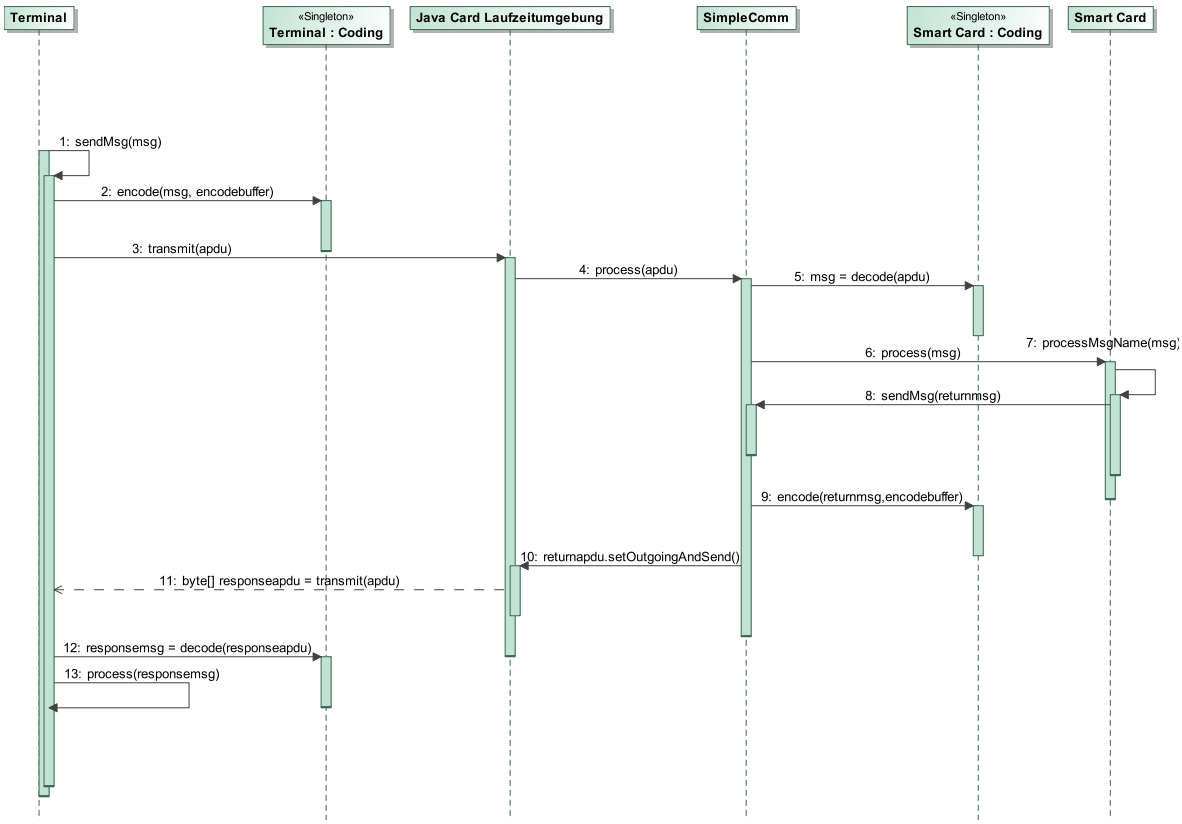

Abbildung 6.4.: Kommunikation zwischen einem Terminal und einer Smart Card

1. Das Terminal ruft, am Ende eines Protokollschritts (vgl. Protokollschritt A in Abb. 6.2 rechts), die Methode `sendMsg` auf, um eine Nachricht an die Karte zu senden. Das zu sendende Nachrichtenobjekt wird als Argument (`msg`) übergeben.
2. Das Nachrichtenobjekt `msg` muss zunächst serialisiert werden. Dies passiert in der Klasse `Coding` in der Methode `encode`. Das Byte-Array `encodebuffer`, in das die serialisierte Nachricht gespeichert werden soll, wird in der Methode `sendMsg` erzeugt und ist so lang, dass jede Nachricht, die in der Anwendung vorkommt, serialisiert werden kann. Diese Maximallänge wird während der Codegenerierung anhand der Klassendiagramme berechnet. Die Methode `encode` serialisiert das Objekt `msg` und speichert das Ergebnis im Array `encbuffer`.
3. Als nächstes wird die APDU erstellt, die an die Karte gesendet wird. Die serialisierte Nachricht ist im Datenbereich der APDU gespeichert. Die APDU wird dann mithilfe der Methode `transmit` an die Karte geschickt.
4. Diese Nachricht wird dort von der Java Card-Laufzeitumgebung empfangen und von ihr an die `process`-Methode der Klasse `SimpleComm` weitergeleitet. Die APDU wird dabei als Argument übergeben.
5. Die `process`-Methode der Klasse `SimpleComm` ist dafür verantwortlich, dass die in der APDU enthaltene serialisierte Nachricht zunächst deserialisiert wird. Hierfür wird die Methode `decode` der Klasse `Coding` aufgerufen, die das Nachrichtenobjekt `msg` zurückgibt.
6. Im Anschluss wird die Methode `process(msg)` der `SmartCard`-Klasse aufgerufen, die die Verarbeitung der Nachricht startet. Hier wird, entsprechend des Typs der Nachricht `msg`, die Methode `processMsgName` aufgerufen, die den eigentlichen Protokollschritt für diese Nachricht durchführt (vgl. Protokollschritt B in Abb. 6.2 rechts). `MsgName` steht für den Namen der Nachrichtenklasse, von dem das Objekt `msg` eine Instanz ist. Diese Methode ist somit abhängig von der konkreten Anwendung. Die Klasse enthält für jede Nachrichtenklasse, deren Instanzen sie verarbeiten kann eine entsprechende Methode, die den Protokollschritt implementiert, der mit dem Empfang der Nachricht beginnt.
7. Die Nachricht wird somit in der Methode `processMsgName(msg)` verarbeitet.

Bisher wurde also eine Nachricht von einem Terminal an eine Smart Card gesendet, die nun den zugehörigen Protokollschritt ausführt. Tritt dabei ein Fehler auf oder ist dies der letzte Protokollschritt in dem Protokoll, sendet die Karte ein entsprechendes Statuswort zurück an das Terminal (einen Fehlercode oder das Statuswort „SW_OK“). Im Fehlerfall wirft das Terminal eine `TerminalException` und beendet so den Protokolllauf. In der Regel antwortet die Karte jedoch mit einer Antwortnachricht. Dies ist wie folgt implementiert:

8. Am Ende des Protokollschritts auf der Karte (Protokollschritt B), d.h. am Ende der Methode `processMsgName`, wird die Methode `sendMsg` der Klasse `SimpleComm`, mit der zu sendenden Nachricht `returnmsg` als Argument, aufgerufen. Diese Methode speichert die Nachricht in dem Feld `outmsg` der Klasse. Damit ist der Aufruf der Methode `processMsgName(msg)` sowie der Aufruf von `process(msg)` beendet und die Methode `process(apdu)` der Klasse `SimpleComm` wird weiter ausgeführt. Diese stellt fest, dass das Feld `outmsg` ein zu sendendes Nachrichtenobjekt enthält.

9. Nun wird diese Nachricht serialisiert. Hierfür wird die Methode `encode(returnmsg,encodebuffer)` der Klasse `Coding` aufgerufen. Diese serialisiert das Nachrichtenobjekt in das dafür vorgesehene Byte-Array `encodebuffer`. Wie auch auf Terminalseite ist dieses lang genug, um die Serialisierung jedes validen Nachrichtenobjektes zu speichern.
10. Anschließend wird in der Methode `process(apdu)` die APDU erstellt und durch Aufruf der Methode `setOutgoingAndSend` die Kontrolle an die Java Card-Laufzeitumgebung übergeben.
11. Diese ist für das eigentliche Versenden der Nachricht zurück an das Terminal verantwortlich. Das Terminal erhält die Antwort-APDU der Karte (`responseapdu`) als Rückgabewert des Methodenaufrufs `transmit` und prüft, ob das Statuswort „SW_OK “ ist.
12. In diesem Fall wird die serialisierte Nachricht durch Aufruf der Methode `decode` deserialisiert.
13. Die empfangene Nachricht wird dann auf Terminalseite verarbeitet (vgl. Protokollschritt C). Dies geschieht, analog zur Karte, durch Aufruf der Methode `process(msg)`, die wiederum die dem Nachrichtentyp entsprechende anwendungsabhängige Methode `processMsgName` aufruft.

Für eine bessere Übersichtlichkeit wurde bei der Beschreibung der Kommunikation ein Detail weggelassen: Ein Terminal kann, je nach Anwendung, mit mehreren Smart Card-Lesegeräten verbunden sein und somit mit verschiedenen Karten kommunizieren. Dies ist im Deploymentdiagramm durch die Angabe verschiedener Ports (den Namen der Enden der gerichteten Kommunikationspfade) gelöst. Sind an einer Anwendung mehrere Smart Cards beteiligt, muss beim Senden einer Nachricht im Aktivitätsdiagramm angegeben werden, über welchen Port diese gesendet werden soll. Aus dieser Angabe ergibt sich dann die Smart Card, an die die Nachricht geschickt wird.

Im generierten Terminal-Code wird eine Klasse `Ports` generiert, die für jeden im Deploymentdiagramm modellierten `CommunicationPath`, an dem das Terminal beteiligt ist, eine Konstante definiert. Sind keine expliziten Portnamen angegeben, wird automatisch ein Standardname generiert. Für die Klasse `CopyingMachine` der Kopierkartenanwendung sieht diese Klasse wie folgt aus:

```
1 public class Ports{
2 public final static int CopyingMachine2Copycard = 0;
3 public final static int CardOwner2CopyingMachine = 0;
4 public final static int CardOwner2DepositMachine = 1;}
```

Listing 6.9: Klasse Ports für die CopyingMachine

Der Kommunikationspfad zwischen der `CopyingMachine` und der `Copycard` ist durch die Konstante `CopyingMachine2Copycard` repräsentiert. Die beiden anderen Konstanten werden für die Kommunikation mit dem Benutzer verwendet.

Im generierten Terminal-Code gibt es ein Array, das `SWTReader`-Objekte speichert. `SWTReader` ist ein Interface, das mit einem Kartenlesegerät kommuniziert und u.a. die Methode `transmit` zum Senden einer ADPU an die Karte zur Verfügung stellt. Bei der Initialisierung des Terminals muss eine Klasse übergeben werden, die dieses Interface implementiert.

Dies kann entweder ein Simulationsobjekt (für die Simulation der echten Smart Card und Java Card-Laufzeitumgebung zu Testzwecken) oder eine Instanz der Klasse `PcscReader` für die Kommunikation mit einem auf einer echten Karte laufenden Applet sein. Listing 6.10 zeigt den entsprechenden Ausschnitt der Klasse `CopyingMachine`.

```
1 ...
2 private SWTReader[] reader = new SWTReader[1];
3
4 public void initReader(SWTReader reader, int port){
5         this.reader[port] = reader;}
6 ...
```

Listing 6.10: Ausschnitt aus der Klasse `CopyingMachine`: SWTReader-Array und dessen Initialisierung

Die Klasse `CopyingMachine` deklariert ein Array vom Typ `SWTReader` der Länge eins, da dieses Terminal mit nur einem Kartenleser (und somit einer Smart Card) verbunden ist. Dieses Array muss vor Inbetriebnahme des Terminals initialisiert werden. Dies sollte zu dem Zeitpunkt geschehen, an dem auch z.B. geheime Schlüssel oder andere Initialdaten auf das Terminal gespielt werden. Die Methode `initReader` muss dabei für jeden Eintrag in dem `SWTReader`-Array aufgerufen werden. Die definierten Konstanten entsprechen dabei dem Index im Array. Mit dem Aufruf `initReader(reader, CopyingMachine2Copycard)` wird somit das Objekt `reader` in dem Array an der Stelle `CopyingMachine2Copycard` gespeichert.

Beim Versenden einer Nachricht wird auf diese Weise anhand des Portnamens das richtige Reader-Objekt ermittelt. Dies erfolgt durch Angabe des Portnamens beim Aufruf der Methode `sendMsg`.

Es kann vorkommen, dass eine serialisierte Nachricht zu lang ist, um sie in einer APDU an eine Smart Card zu senden. Der Datenteil einer APDU, die an eine Smart Card geschickt wird, darf bei den in dieser Arbeit verwendeten Smart Cards von Giesecke und Devrient [185] maximal 262 Bytes lang sein (dies ist von der verwendeten Java Card-Laufzeitumgebung abhängig), die Daten in der Antwortnachricht dürfen maximal 256 Byte lang sein. Aus diesem Grund wird zusätzlich automatisch ein Mechanismus generiert, der zu lange Nachrichten in mehrere APDUs aufteilt, diese einzeln sendet und auf der Smart Card wieder zu einem Nachrichtenobjekt zusammensetzt.

Um den generierten Anwendungscode leicht und einfach testen zu können, werden außerdem entsprechende Methoden für die Interaktion mit den Benutzern generiert.

6.2.4. Serialisierung und Deserialisierung von Objekten

Für die APDU-basierte Kommunikation zwischen Terminals und Smart Cards sowie für einige kryptographische Operationen ist es notwendig, Objekte in Byte-Arrays zu serialisieren und später wieder zu deserialisieren. In diesem Abschnitt wird die Funktionsweise der hierfür generierten Serialisierungsschicht beschrieben.

Die (De-)Serialisierung von Objekten ist ein sehr schönes Beispiel für die automatische Generierung von Code. Die (De-)Serialisierung ist für sich genommen nicht sehr kompliziert,

aber sehr schwierig von Hand zu implementieren, ohne dabei (Copy-und-Paste-)Fehler zu machen. Die (De-)Serialisierung ist anwendungsspezifisch, d.h. für jede Anwendung wird anderer Code, abhängig von den modellierten Klassen und ihren Attributen und Assoziationen, benötigt. Diese Informationen sind in den Klassendiagrammen enthalten. Somit kann der komplette Code für die (De-)Serialisierung automatisch generiert werden.

Die Serialisierung verwendet eine Typ-Length-Value (TLV) Kodierung, die von den ASN1-Kodierungsregeln für beliebige Daten inspiriert ist [52]. Die von SecureMDD verwendete Kodierung beginnt mit einem ein Byte langen Typflag. Diesem folgt in zwei Bytes die Länge der kodierten Daten. Die Längenangabe wird nur dann mitkodiert, wenn die Länge der Daten variabel ist. Bei der Kodierung z.B. eines Short-Wertes, der zwei Bytes lang ist, ist keine Längenangabe erforderlich. Der Längenangabe folgt die Kodierung der eigentlichen Daten.

6.2.4.1. Definition der Typflags: Klasse `Code`

Die Typflags sind in einer Java-Klasse mit Namen `Code` definiert.

```
1  public class Code {
2          public final static byte IGNORE = (byte) 1;
3          public final static byte BOOLEAN = (byte) 2;
4          public final static byte BYTE = (byte) 3;
5          public final static byte INT = (byte) 4;
6          public final static byte BOOLEANARRAY = (byte) 5;
7          public final static byte BYTEARRAY = (byte) 6;
8          public final static byte SHORTARRAY = (byte) 7;
9          public final static byte NONCE = (byte) 8;
10         public final static byte SECRET = (byte) 9;
11         public final static byte SYMMKEY = (byte) 10;
12         public final static byte PRIVATEKEY = (byte) 11;
13         public final static byte PUBLICKEY = (byte) 12;
14         public final static byte HASHEDDATA = (byte) 13;
15         public final static byte SIGNEDDATA = (byte) 14;
16         public final static byte ENCDATASYMM = (byte) 15;
17         public final static byte ENCDATAASYMM = (byte) 16;
18         public final static byte MACDATA = (byte) 17;
19         public final static byte AUTHDATA = (byte) 18;
20         public final static byte AUTHENTICATE = (byte) 19;
21         public final static byte COPYCARD = (byte) 20;
22         public final static byte LOAD = (byte) 21;
23         public final static byte MESSAGE = (byte) 22;
24         public final static byte PAY = (byte) 23;
25         public final static byte REQUESTBALANCE = (byte) 24;
26         public final static byte RESAUTHENTICATE = (byte) 25;
27         public final static byte RESLOAD = (byte) 26;
28         public final static byte RESPAY = (byte) 27;
29         public final static byte RESREQUESTBALANCE = (byte) 28;}
```

Listing 6.11: Klasse `Code` mit Typflags am Beispiel der Kopierkartenanwendung

Diese Klasse enthält eine Konstante vom Typ Byte für alle Daten- und Nachrichtenklassen, deren Instanzen während der Ausführung der Protokolle serialisiert werden müssen. Listing 6.11 zeigt beispielhaft die Klasse `Code` für die Kopierkartenanwendung. Die Klasse ist für die Terminal- und die Smart Card-Komponente gleich. Das Typflag `IGNORE` wird für die Initialisierung einer Smart Card benötigt. Die nachfolgenden Konstanten (`BOOLEAN` bis `MACDATA`) definieren die Typflags für die primitiven Typen sowie die vordefinierten Sicherheitsdatentypen. Bis zu dieser Stelle werden für jede Anwendung die gleichen Flags generiert. Es folgen nun noch die anwendungsabhängigen Typflags. In der Kopierkartenanwendung sind dies die Nachrichtenklassen sowie die Klasse `AuthData`, die von zwei Nachrichtenklassen assoziiert wird. Außerdem wird ein Typflag für die Klasse `Copycard` (d.h. dem Karten-Applet) generiert. Dieses wird für die Initialisierung der Karte verwendet.

Um auszudrücken, dass Objekte einer Klasse zur Laufzeit (de-)serialisiert werden können, implementiert diese Klasse das Interface `Codeable`, siehe Listing 6.12. Das Interface deklariert die Methode `getCode()`, die das Typflag der Klasse zurückgibt. Das Interface wird bei der Generierung des Java-Codes unabhängig von der konkreten Anwendung erstellt.

```
public interface Codeable {
public byte getCode(); }
```

Listing 6.12: Interface `Codeable`

Die Festlegung, welche Klassen „kodierbar sind“, d.h. das Interface implementieren, wird bei deren Generierung automatisch anhand der plattformunabhängigen Klassendiagramme bestimmt. Kodierbar sind:

1. alle Nachrichtenklassen, da die Nachrichtenobjekte beim Versenden serialisiert werden müssen.
2. alle Klassen mit Stereotyp ≪PlainData≫, ≪SignData≫ oder ≪HashData≫. Instanzen dieser Klassen werden während der Protokollausführung verschlüsselt, signiert oder es wird der Hash- oder MAC-Wert gebildet. Die in der Implementierung verwendeten API-Methoden zum Verschlüsseln, Signieren und Hashen basieren auf Byte-Arrays. Die entsprechenden Eingabeargumente für diese Methoden sind deshalb ebenfalls kodierte Objekte.
3. Klassen, die für die Initialisierung einer Smart Card relevant sind. Dies sind Klassen, die von einer Smart Card-Klasse assoziiert werden und deren gerichtetes Assoziationsende mit dem Stereotyp ≪Initialize≫ annotiert ist (siehe Abschnitt 4.2.1.9). Die Initialisierung erfolgt ebenfalls durch Senden einer entsprechenden APDU an die Karte. Folglich müssen auch diese Klassen kodierbar sein.
4. Alle Datenklassen, die von den unter 1 bis 3 genannten Klassen assoziiert werden. Hierzu zählen auch Datenklassen, die indirekt (durch Bilden der transitiven Hülle) assoziiert werden. Der Grund hierfür ist, dass bei Serialisierung der unter 1 bis 3 genannten Instanzen auch die entsprechenden Unterobjekte serialisiert werden müssen. Die Serialisierung erfolgt deshalb rekursiv über die Attribute und Assoziationsenden einer zu serialisierenden Klasse.

6.2.4.2. Kodierung der Daten: Klasse `Coding`

Die eigentliche Serialisierung und Deserialisierung der Daten findet in der Klasse `Coding` statt. Das prinzipielle Vorgehen der (De-)Serialisierung ist natürlich für die Terminals sowie die Smart Cards gleich. Dennoch gibt es bei der Implementierung der Klasse `Coding` einige Unterschiede. Der gravierendste Unterschied ist die Verwendung des Objektmanagers, der die Objekte, in die die serialisierten Daten dekodiert werden, bereitstellt. Im Terminal-Code hingegen werden diese Objekte bei Bedarf neu erzeugt. Das gleiche gilt für die Verwendung einiger Byte-Arrays. Im Java Card-Code werden diese bei der Erzeugung des Applets erstellt und dann wiederverwendet, während auf Terminalseite die Byte-Arrays dann erzeugt werden, wenn sie benötigt werden. Im Folgenden werden die Klasse `Coding` bzw. einige ihrer Methoden zum Serialisieren und Deserialisieren anhand der Kopierkartenanwendung im Detail vorgestellt. Es wird die Implementierung für die Smart Card, d.h. für die Komponente `Copycard`, vorgestellt und die Unterschiede zum Terminal-Code, falls nötig, erläutert.

Serialisierung von Objekten Die Klasse `Coding` deklariert eine Methode `encode(Codeable o, byte[] dest)` zum Serialisieren von Objekten. Diese wird aufgerufen, wenn ein Objekt (vom Typ `Codeable`) serialisiert werden soll. Die Methode ist in Listing 6.13 abgebildet. Ein Beispiel für den Aufruf der Methode ist in Abbildung 6.4 des vorherigen Abschnitts zu sehen.

```
1 public short encode(Codeable o, byte[] dest) {
2         encodeInit(dest);
3         encode(o);
4         return encoding_length; }
5
6 public void encodeInit(byte[] dest) {
7         encodeDestination = dest;
8         encoding_length = 0; }
```

Listing 6.13: Methode `encode(Codeable c, byte[] dest)` Klasse `Coding`

Die Methode `encode` bekommt das zu serialisierende Objekt `o` sowie das Byte-Array `dest`, in welches das Objekt serialisiert werden soll, übergeben. Zunächst wird dieses Array in der Variablen `encodeDestination` gespeichert und die Variable `encoding_length` vom Typ `short` auf 0 gesetzt. Diese Variable gibt an, wie viele Bytes bereits serialisiert wurden. Im Anschluss wird die Methode `encode(Codeable c)` aufgerufen, die die eigentliche Serialisierung durchführt. Die Methode gibt die Länge des serialisierten Objekts zurück.

Die Methode `encode(Codeable c)` der Kopierkartenanwendung ist in Listing 6.14 abgebildet.

Die Methode ist abhängig von der konkreten Anwendung sowie der Komponentenklasse, für die sie generiert wird. In diesem Beispiel ist dies die Kopierkarte, die hier die einzige Smart Card-Komponente ist. Wären im UML-Modell mehrere Smart Card-Komponentenklassen modelliert, würde für jede eine eigene `Coding`-Klasse generiert werden. Die Methode `encode` enthält eine `switch`-Anweisung, die anhand des Typflags (`c.getCode()`, Zeile 2) verzweigt und die entsprechende `encode`-Methode des Objekts aufruft. Die Codegenerierung ermittelt anhand des UML-Modells, welche der als kodierbar errechneten Klassen für die entsprechen-

```
1  public void encode(Codeable c) {
2  switch (c.getCode()) {
3          case Code.LOAD :
4                  encodeLoad((Load) c);
5                  break;
6          ...
7          case Code.HASHEDDATA :
8                  encodeHashedData((HashedData) c);
9                  break;
10         case Code.AUTHDATA :
11                 encodeAuthData((AuthData) c);
12                 break;
13         case Code.NONCE :
14                 encodeNonce((Nonce) c);
15                 break;
16         case Code.SECRET :
17                 encodeSecret((Secret) c);
18                 break;
19         default :
20                 SimpleComm.stop();}}
```

Listing 6.14: Methode `encode(Codeable c)` der Klasse `Coding`

de Komponentenklasse relevant sind und generiert nur für diese ein Sprungziel (`case`). Für die Komponentenklasse `Copycard` sind dies alle Nachrichtenklassen bzw. -objekte, die an die Kopierkarte gesendet bzw. von ihr versendet werden. Dies ist beispielhaft für die Klasse `Load` angegeben (Zeile 3 - 5), der Code für die restlichen Nachrichtenklassen funktioniert analog („...“ im Listing). Außerdem kodiert werden muss die Klasse `HashedData`, da die Nachrichtenklassen `Load` und `Respay` einen Hashwert enthalten. Weiterhin kodiert werden Instanzen der Klasse `AuthData` (da sie mit ≪HashData≫ annotiert ist), sowie Instanzen der Klassen `Nonce` und `Secret` (da sie Attribute der Klasse `AuthData` sind und somit vor Bildung des Hashwertes ebenfalls serialisiert werden müssen). Die Default-Marke (Zeile 19) wird erreicht, wenn ein Objekt `c` keines der angegebenen Typflags besitzt. Da der generierte Code diese Methode jedoch nur für Objekte aufruft, die auch kodierbar sind, wird dieser Default-Zweig nie ausgeführt. Er ruft die Methode `stop()` auf, die eine ISOException wirft.

Die Serialisierung der Klassen ist rekursiv über die Felder der Klasse implementiert. Dies wird beispielhaft anhand der Serialisierung der Nachrichtenklasse `Load` erläutert. Der Aufruf der Methode `encodeLoad` serialisert nacheinander die Felder `amount` und `authterminal` der Klasse und ruft dabei rekursiv die entsprechenden `encode`-Methoden auf. Die TLV-Kodierung der Klasse `Load` ist in Abbildung 6.5 dargestellt.

Die Serialisierung beginnt mit dem Typflag der `Load`-Klasse (LOAD), gefolgt von der Serialisierung des Feldes `amount` vom Typ `short`. In SecureMDD werden Shortwerte (sowie Integer auf Terminalseite) als Integer kodiert. Der Grund hierfür ist, dass auf Terminalseite der abstrakte Typ `Number` aus dem UML-Modell in einen Integer übersetzt wird, während auf der Smart Card Short-Werte verwendet werden. Die Kodierung fasst beide Typen zusammen und kodiert auch Short-Werte als Integer. Die Serialisierung eines Short-Wertes besteht

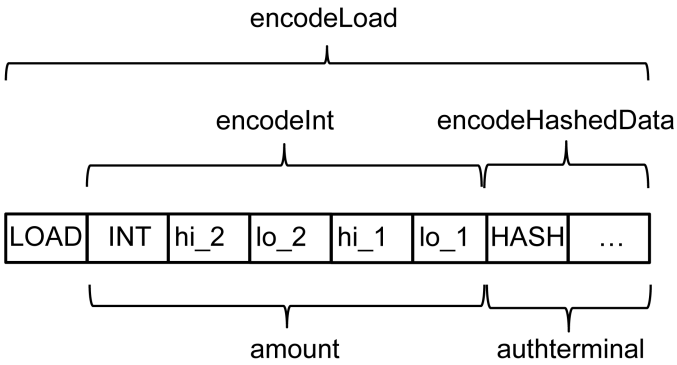

Abbildung 6.5.: Serialisierung der `Load`-Klasse

somit aus dem Typflag für Integer (INT), gefolgt von den vier Datenbytes (`hi_2` - `lo_1`). Im Falle eines Shorts sind die beiden höheren Bytes (`hi_2` und `lo_2`) gleich 0. Anschließend folgt die Kodierung des Hashwertes, die das Typflag HASH verwendet und anschließend den eigentlichen Hashwert serialisiert.

Die entsprechende Implementierung der Methode `encodeLoad` ist in Listing 6.15 angegeben.

```
1 private void encodeLoad(Load c) {
2         encodeDestination[encoding_length++] = Code.LOAD;
3         encodeInt((short) c.amount);
4         encode(c.authterminal);}
```

Listing 6.15: Methode `encodeLoad` für die Serialisierung eines `Load`-Objekts auf der Kopierkarte

In Zeile 2 wird das Typflag für die Nachricht `Load` in das Byte-Array `encodeDestination` geschrieben und Variable `encoding_length` inkrementiert. Anschließend wird durch Aufruf der Methode `encodeInt` der Short-Wert `amount` und durch Aufruf der Methode `encode(Codeable c)` (siehe List. 6.14) der Hashwert `authterminal` vom Typ `HashedData` serialisiert.

Diese beiden Methoden sind im Folgenden erläutert. Listing 6.16 zeigt den Quellcode der Methode `encodeInt`.

```
1 private void encodeInt(short o) {
2 encodeDestination[encoding_length++] = Code.INT;
3 Util.setShort(encodeDestination,encoding_length,(short)0x00);
4 Util.setShort(encodeDestination,(short)(encoding_length + 2),o);
5 encoding_length += 4; }
```

Listing 6.16: Methode `encodeInt` für die Serialisierung eines Short-Wertes auf der Kopierkarte

Nach dem Schreiben des Typflags INT in das Array `encodeDestination` (Zeile 2), wird der eigentliche Wert serialisiert. Dies geschieht durch Setzen der oberen beiden Bytes auf 0 (Zeile

3) und dem anschließenden Schreiben des Short-Wertes in die zwei folgenden Bytes. Zuletzt wird noch die Variable `encoding_length` um die vier geschriebenen Bytes erhöht (Zeile 5). Im Terminal-Code ist die Methode äquivalent implementiert. Der einzige Unterschied ist, dass dort ein Integer in das Byte-Array geschrieben wird anstatt eines Short-Wertes.

Das Ergebnis der Operation zum Hashen eines Datums (das als Byte-Array vorliegen muss), ist wieder ein Byte-Array. Aus diesem Grund besitzt der Datentyp `HashedData` ein Feld `hashed` vom Typ Byte-Array, in dem der Hashwert gespeichert ist. Die Serialisierung eines Objekts vom Typ `HashedData` besteht deshalb aus dem Typflag `HASH`, gefolgt von der Serialisierung des Byte-Arrays, das den Hashwert speichert.

Die Serialisierung eines Byte-Arrays ist in Abbildung 6.6 dargestellt.

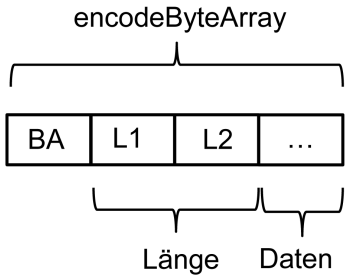

Abbildung 6.6.: Serialisierung eines Byte-Arrays auf der Smart Card

Sie besteht aus dem Typflag für das Byte-Array, gefolgt von zwei Bytes für die Länge des Arrays sowie dem eigentlichen Array. Listing 6.17 zeigt die Implementierung der Methode `encodeByteArray` für eine Smart Card-Komponente.

```
1 public void encodeByteArray(byte[] o) {
2 encodeDestination[encoding_length++] = Code.BYTEARRAY;
3 Util.setShort(encodeDestination, encoding_length,
4         (short) (o.length));
5 encoding_length += 2;
6 Util.arrayCopy(o, (short) 0, encodeDestination,
7         encoding_length, (short) o.length);
8 encoding_length += o.length; }
```

Listing 6.17: Methode `encodeByteArray` für die Serialisierung eines Byte-Arrays auf der Kopierkarte

Zunächst werden das Typflag (Zeile 2) und die Länge des Arrays (Zeile 3-4) gesetzt. Anschließend wird die Variable `encoding_length` um die zwei Bytes für die geschriebenen Daten erhöht (Zeile 5) und das Byte-Array `o` ab Index 0 in das Ziel-Array ab Index `encoding_length` kopiert (Zeilen 6-7). Anschließend wird die `encoding_length` entsprechend der Länge des kopierten Arrays erhöht (Zeile 8).

Nach Kodierung des in dem Hashwert der `Load`-Nachricht enthaltenen Byte-Arrays ist die Serialisierung dieser Nachricht abgeschlossen. Die Serialisierung der weiteren Nachrichten- und Datenklassen erfolgt auf analoge Weise. Die Sicherheitsdatentypen `Secret`, `Nonce`, `SymmKey`

und SignedData speichern, wie auch der Datentyp HashedData, ein Byte-Array und werden auf die gleiche Weise serialisiert wie dieser. Bei öffentlichen und privaten Schlüsseln werden der Modulus bzw. Modulus und Exponent (die als Byte-Array gespeichert werden) nacheinander serialisiert. Eine Ausnahme bilden die Datentypen EncDataSymm sowie EncDataAsymm, bei denen zusätzlich zu dem eigentlichen Byte-Array encrypted, das die verschlüsselten Daten enthält, noch weitere Informationen in dem serialisierten Array gespeichert werden. Dies sind die Länge der (serialisierten) Klartextdaten (kodiert als Integer), die Länge der verschlüsselten Daten (ebenfalls als Integer kodiert) sowie der Typ der Klartextdaten (kodiert als Byte). Diese Informationen sind notwendig, um nach dem Deserialisieren und vor dem Entschlüsseln des Arrays encrypted zu überprüfen, ob das verschlüsselte Datum valide ist, d.h. beim Aufruf der API-Methode zum Entschlüssen keine Java RuntimeException geworfen wird.

Eine weitere Ausnahme bildet die Kodierung von Strings im Terminal-Code. Der in UML definierte, primitive Typ String wird bei der Generierung des Terminal-Codes in den Java-Typ String übersetzt. Da dieser Typ von Java Card nicht unterstützt wird, ist ein String im Smart Card-Code durch ein Byte-Array repräsentiert. Die Methode encodeString wird deshalb nur auf Terminalseite unterstützt. Diese Methode übersetzt den zu serialisierenden String zunächst in ein Byte-Array und serialisiert dieses dann durch Aufruf der Methode encodeByteArray. Für die Übersetzung des Strings in ein Byte-Array wird die ASCII-Kodierung des Strings verwendet. Hierfür wird die Methode getBytes der Java-Klasse String aufgerufen.

Deserialisierung von Objekten Die Implementierung der Deserialisierung für die Terminals unterscheidet sich von der für die Smart Cards darin, dass die Objekte, in die die Daten deserialisiert werden sollen, auf Terminalseite bei Bedarf erzeugt werden können. Auf einer Smart Card ist dies aufgrund der begrenzten Speicherkapazität nicht möglich (siehe Abschnitt 6.2.5). Hier werden die benötigten Objekte von dem Objektmanager zur Verfügung gestellt. Darüber hinaus sind beide Implementierungen, wie auch bei der Serialisierung, sehr ähnlich. Im Folgenden wird deshalb wieder der generierte Code für eine Smart Card-Komponente betrachtet. Gibt es Unterschiede zwischen dem Smart Card- und dem Terminal-Code, wird darauf explizit hingewiesen.

Analog zu der Erläuterung der Serialisierung wird der Empfang und das anschließende Deserialisieren eines Objekts der Nachrichtenklasse Load auf der Kopierkarte betrachtet. Hierfür wird beim Empfang der Nachricht von der Kartenklasse SimpleComm die Methode decodeMessage der Klasse Coding aufgerufen. Diese ist in Listing 6.18 abgebildet.

Die Methode bekommt als Argumente das Byte-Array mit den kodierten Daten, den Index für den Zugriff auf das erste Byte der Kodierung (offset) sowie die Länge der kodierten Daten (expectedLength) übergeben. Die Switch-Anweisung verzweigt anhand des Typflags in dem Byte-Array, das den Typ der Nachricht angibt. Für jeden Nachrichtentyp, der von der Karte dekodiert werden kann, wird ein Case-Zweig generiert („... “ in Zeile 11). Der Case-Zweig für den Typ LOAD holt sich zunächst ein Objekt vom Typ Load vom Objektmanager (Zeile 8), in das die kodierten Daten dann deserialisiert werden. Dies geschieht durch Aufruf der Methode decodeLoadInto. Nach der Deserialisierung der Nachricht wird überprüft, ob die Anzahl der Bytes, die deserialisiert wurden, mit der erwarteten Anzahl (d.h. den empfangenen Bytes) übereinstimmt. Ist dies nicht der Fall, deutet dies darauf hin, dass ein

```
1  public Message decodeMessage(byte[] in,short offset,
2        short expectedLength) {
3  encoding_length = offset;
4  Message m = null;
5  try{
6     switch (in[encoding_length]) {
7        case Code.LOAD :
8           m = Store.newLoad();
9           decodeLoadInto(in, (Load) m);
10          break;
11       ... }}
12 catch(ArrayIndexOutOfBoundsException e){
13    SimpleComm.stop();}
14
15 if (m == null || encoding_length !=
16       (short)(expectedLength + offset)) {
17    SimpleComm.stop();}
18 return m;}
```

Listing 6.18: Methode `decodeMessage` für die Deserialisierung einer Nachricht auf der Kopierkarte

Angreifer das gesendete Byte-Array manipuliert hat (siehe Abschnitt 10.3.6). Damit es an dieser Stelle nicht zu Sicherheitslücken kommt, wird die Methode `stop()` aufgerufen, die den Protokollschritt abbricht und eine Exception wirft. Im Nicht-Fehlerfall gibt die Methode das Nachrichtenobjekt zurück an den Aufrufer.

Die Methode `decodeLoadInto(in,(Load)m)` ist in Listing 6.19 abgebildet. Dort wird zunächst geprüft, ob das Typflag dem einer `Load`-Nachricht entspricht. Anschließend werden die beiden Felder `amount` und `authterminal` deserialisiert.

```
1 private void decodeLoadInto(byte[] in, Load into) {
2 if (in[encoding_length++] != Code.LOAD)
3    SimpleComm.stop();
4 into.amount = decodeInt(in);
5 decodeHashedDataInto(in,(HashedData)(into.authterminal)); }
```

Listing 6.19: Methode `decodeLoadInto` für die Deserialisierung einer `Load`-Nachricht auf der Kopierkarte

In Zeile 4 wird die Methode `decodeInt` aufgerufen, die auf einer Smart Card einen Short-Wert zurückgibt. Dieser wird in dem Feld `amount` des Objekts `into` gespeichert. Anschließend wird der in der Nachricht enthaltene Hashwert deserialisiert und in dem Feld `authterminal` gespeichert (Zeile 5).

Die Implementierung der Methode `decodeInt` ist für Terminals und Smart Cards unterschiedlich. Der Grund hierfür ist, dass der in UML verwendete primitive Typ `Number` im Terminal-Code in Integer, im Smart Card-Code jedoch in den Typ `short` übersetzt wird. Beide Typen werden jedoch als Integer kodiert. Auf Terminalseite wird ein kodierter Inte-

ger beim Dekodieren als Integer gespeichert, auf Smart Card-Seite findet ein Cast auf einen `short`-Wert statt. Dabei kann es passieren, dass der kodierte Integer zu groß bzw. zu klein ist, um ihn als `short`-Wert zu speichern. In diesem Fall wirft die Methode `decodeInt` eine Exception und beendet somit den Protokollschritt. Die Begründung für dieses Verhalten ist, dass der Überlauf des Wertebereichs vermutlich von dem Entwickler der Anwendung nicht intendiert ist und somit unbehandelt eventuell ein Sicherheitsproblem darstellt. Die Implementierung der Methode `decodeInt` ist Listing 6.20 dargestellt.

```
1  private short decodeInt(byte[] in) {
2  if (in[encoding_length++] != Code.INT){
3     SimpleComm.stop();}
4  short upper = Util.getShort(in, encoding_length);
5  if (upper != 0x00) {
6     SimpleComm.stop();
7     return 0; }
8  else {
9     short lower = Util.getShort(in,(short)(encoding_length + 2));
10    encoding_length += 4;
11    return lower; } }
```

Listing 6.20: Methode `decodeInt` für die Deserialisierung eines Integers bzw. short-Wertes auf der Kopierkarte

In Zeile 4 werden die beiden höheren Bytes des kodierten Integers in der Variablen `upper` gespeichert. Ist dieser Short-Wert ungleich 0, ist der Integer zu groß, um ihn als Short-Wert zu speichern (Zeile 5). In diesem Fall, wird die Methode `stop()` der Klasse `SimpleComm` aufgerufen, die eine Exception wirft. Zeile 7, die Rückgabe des Wertes 0, wird deshalb nie ausgeführt. Passt der Integer in eine Variable vom Typ `short`, werden die beiden unteren Bytes aus dem Array gelesen und in der Variablen `lower` gespeichert (Zeile 9). Die Zahl der bereits dekodierten Bytes wird um vier erhöht (Zeile 10) und der Wert `lower` zurück an den Aufrufer gegeben.

Die Dekodierung des Hashwertes ist in der Methode `decodeHashedDataInto(byte[] in, HashedData into)` implementiert. Sie prüft zunächst, ob das Typflag in dem Byte-Array `in` gleich dem Flag `HASHEDDATA` ist und ruft dann die Methode `decodeByteArrayInto` auf, die als Argumente das Byte-Array `in` und das Feld `hashed` des Objekts `into` übergeben bekommt.

Die Methode `decodeByteArrayInto` ist in Listing 6.21 abgebildet.

Ist das Typflag korrekt (Test in Zeile 2), werden die zwei Bytes, die die Länge des kodierten Byte-Arrays speichern, aus dem Array `in` gelesen und in der Variablen `len` gespeichert (Zeile 4). Anschließend wird die `encoding_length` um die zwei gelesenen Längenbytes erhöht und überprüft, ob der Wert der Variablen `len` valide ist (damit beim Kopieren keine `ArrayIndexOutOfBoundsException` auftreten kann, Zeilen 6-7). Schlägt der Test fehl, deutet dies darauf hin, dass ein Angreifer die Daten bei der Übertragung manipuliert hat (siehe Abschnitt 10.3.6). In diesem Fall wird die Methode `stop()` aufgerufen, die eine Exception wirft. Ist der Test erfolgreich, wird das kodierte Byte-Array aus dem Array `in` in das Byte-Array `into` kopiert (Zeile 9). Ist das Byte-Array `into` länger als das kodierte Byte-Array, werden die restlichen Bytes des Arrays `into` mit Nullen aufgefüllt (Zeilen 10 bis 12). Ab-

```
1  private void decodeByteArrayInto(byte[] in, byte[] into) {
2  if (in[encoding_length++] != Code.BYTEARRAY)
3     SimpleComm.stop();
4  short len = Util.getShort(in, encoding_length);
5  encoding_length += 2;
6  if (!(0 <= len && len <= into.length &&
7        len <= (in.length - encoding_length)))
8     SimpleComm.stop();
9  Util.arrayCopy(in, encoding_length, into, (short) 0, len);
10 if (len < into.length)
11    Util.arrayFillNonAtomic(into,len,((short)(into.length-len)),
12          (byte) 0);
13 encoding_length += len; }
```

Listing 6.21: Methode `decodeByteArrayInto` für die Deserialisierung eines Byte-Arrays auf der Kopierkarte

schließend wird die Variable `enconding_length` entsprechend der kopierten Bytes erhöht (Zeile 13).

Die Deserialisierung der weiteren kodierbaren Klassen und primitiven Typen ist analog implementiert. Lediglich die Deserialisierung von verschlüsselten Daten (d.h. der Klassen `EncDataSymm` und `EncDataAsymm`) ist, wie auch bei der Serialisierung, gesondert zu betrachen. Die entsprechenden `decode`-Methoden dekodieren nach dem eigentlichen Byte-Array mit den verschlüsselten Daten noch die Klartextlänge und speichern sie in dem Feld `plainlength` der Klasse `EncDataSymm` bzw. `EncDataAsymm`. Anschließend werden noch die Länge der verschlüsselten Daten sowie der Datentyp des Klartextobjekts dekodiert und in den Feldern `enclength` und `plaindatatype` gespeichert. Ist der in `plaindatatype` gespeicherte Typ nicht verschlüsselbar, d.h. implementiert die zu dem Typflag gehörende Klasse nicht das Interface `PlainData`, wird die `stop()`-Methode aufgerufen, die eine Exception wirft. In diesem Fall hat der Angreifer das Byte-Array manipuliert. Die Länge `plainlength` und `enclength` werden nicht beim Dekodieren, sondern erst beim (späteren) Entschlüsseln der Daten überprüft.

6.2.5. Objektverwaltung auf der Smart Card durch einen Objektmanager

Aufgrund des geringen Speicherplatzes auf einer Smart Card und der fehlenden Garbage Collection, können auf der Smart Card nicht beliebig viel Speicher allokiert und somit Objekte erzeugt werden. Aus diesem Grund werden alle während der Protokollläufe auf den Smart Cards benötigten Objekte bereits während der Erzeugung der Appletklasse erzeugt (in dem Konstruktor der Klasse `SimpleComm`). Die benötigten Objekte lassen sich unterteilen in

- die Objekte für die Felder der Appletklasse und ihre Unterobjekte. Wird während eines Protokollschritts ein neues Objekt an ein Feld zugewiesen, werden die neuen (Objekt-) Werte der Felder des zugewiesenen Objekts kopiert, anstatt, wie in Java üblich, die Referenz auf das zugewiesene Objekt zu speichern. Auf diese Weise bleibt das einmal an das Feld zugewiesene Objekt immer das gleiche, nur die Werte der Felder ändern sich.

- die Objekte, die für die lokalen Variablen innerhalb der Protokollschritte benötigt werden und ihre Unterobjekte.
- die Nachrichtenobjekte, in die die kodierten Nachrichten nach dem Empfang deserialisiert werden und ihre Unterobjekte.

Die für die lokalen Variablen und Nachrichten benötigten Objekte werden während des Erzeugens der Appletklasse von einem Objektmanager erstellt und verwaltet. Wird während eines Protokollschritts ein Objekt benötigt, gibt der Objektmanager dieses heraus. Da die lokalen Variablen sowie die Nachrichtenobjekte nur bis zum Ende des Protokollschritts gültig sind, können alle in dem Schritt verwendeten Objekte am Ende des Schrittes an den Objektmanager „zurückgegeben“ werden.

Der Objektmanager muss ausreichend Objekte zur Verfügung stellen, damit jeder Protokollschritt, der auf der Smart Card stattfindet, zu jedem Zeitpunkt ausführbar ist. Dies muss deshalb gewährleistet sein, weil ein Angreifer (mit entsprechenden Fähigkeiten) eine beliebige Nachricht an die Karte schicken kann, die dann dort verarbeitet wird. Welche Klassen von dem Objektmanager verwaltet werden und wie viele Instanzen dieser jeweils vorhalten muss, wird während der Codegenerierung automatisch anhand der Aktivitätsdiagramme errechnet. Für jede Nachrichtenklasse, die von der Karte empfangen oder versendet wird, muss ein Objekt im Objektmanager verwaltet werden. Für jeden weiteren Datentyp wird für jeden Protokollschritt der Karte berechnet, wie viele lokale Variablen dieses Typs in diesem Schritt deklariert werden und anschließend das Maximum über die verschiedenen Werte genommen. Zusätzlich ist bei dieser Berechnung berücksichtigt, dass die Implementierung einiger vordefinierter Methoden ebenfalls Objekte vom Objektmanager anfordern. Zum Beispiel erzeugt die Methode `generateNonce()` ein neues Objekt vom Typ Nonce, das vom Objektmanager zur Verfügung gestellt wird.

Im Folgenden ist die Implementierung des Objektmanagers am Beispiel der Nachrichtenklasse `Load` angegeben. Die Verwaltung aller weiteren Objekte ist analog implementiert. Listing 6.22 zeigt den entsprechenden Ausschnitt der Klasse `Store`, die den Objektmanager implementiert.

```
1  private static Load[] LoadArray;
2  private static byte LoadCount = 0;
3
4  private static void initLoad() {
5          LoadArray = new Load[LoadMaxCountForCopycard];
6          for (short i = 0; i < LoadArray.length; i++)
7                  LoadArray[i] = new Load();}
8
9  public static Load newLoad() {
10         return LoadArray[LoadCount++]; }
```

Listing 6.22: Ausschnitt aus dem Objektmanager für die Objekte der Klasse `Load`

Das Array `LoadArray` speichert die verwalteten `Load`-Objekte, die Variable `LoadCount` speichert den Index für das nächste, noch nicht verwendete `Load`-Objekt. Die Methode `initLoad()` wird bei Erzeugung des Applets aufgerufen. Sie erzeugt ein Array der Länge `LoadMaxCountForCopycard` (Zeile 5). Die Konstante `LoadMaxCountForCopycard`

gibt an, wie viele `Load`-Objekte vom Objektmanager verwaltet werden müssen (in diesem Fall eins). Anschließend werden die `Load`-Objekte erzeugt und in dem Array gespeichert. Dabei werden gleichzeitig auch alle Unterobjekte erzeugt (in diesem Fall ist dies ein Objekt vom Typ `HashedData` für das Feld `authterminal`). Wird während eines Protokollschritts ein `Load`-Objekt benötigt, wird die Methode `newLoad()` des Objektmanagers aufgerufen, der das nächste freie Objekt zurückgibt und den Index `LoadCount` inkrementiert (Zeilen 10 bis 11). Am Ende eines Protokollschritts werden alle Objekte an den Objektmanager „zurückgegeben“. Realisiert ist dies dadurch, dass die Variablen, die den Index auf das nächste freie Objekt speichern (`LoadCount` für die `Load`-Objekte), zurück auf 0 gesetzt werden. Dadurch, dass bei Zuweisungen die entsprechenden Werte der Objekte immer kopiert werden, ist sichergestellt, dass die zurückgegebenen Objekte nicht mehr referenziert werden.

6.3. Generierung des Codes

In diesem Abschnitt sind (informell) die Transformationsregeln beschrieben, die aus einem plattformunabhängigen UML-Modell den Quellcode für Smart Card und Terminals erzeugen. Die Implementierung der Transformationen ist in [129] veröffentlicht.

Die Generierung des Codes ist an einigen Stellen anhand der Kopierkartenanwendung erläutert und bezieht sich aus das in den Abschnitten 4.3 und 4.5 vorgestellte UML-Modell der Anwendung. In der Regel wird der Smart Card-Code abgebildet und auf Unterschiede zum Terminal-Code hingewiesen.

Allgemein ist zu sagen, dass für jede Komponentenklasse (d.h. eine Klasse mit Stereotyp ≪Smartcard≫ oder ≪Terminal≫) ein eigenes Java-Package erzeugt wird, das alle generierten Klassen für diese Komponente enthält. Auf diese Weise ist es leicht möglich, den Code später auf einem (physikalischen) Terminal oder einer Smart Card zu deployen. Die mit Stereotyp ≪User≫ annotierten Benutzerklassen, die die Benutzer der Anwendung repräsentieren, werden nicht in Code übersetzt.

6.3.1. Transformation der Klassendiagramme in Code

Tabelle 6.1 listet die in den Klassendiagrammen verwendeten UML-Elemente und ihre Übersetzung in Code auf. Die zweite Spalte der Tabelle bezieht sich auf die Modellierung mit UML. In der dritten Spalte ist die Umsetzung im Terminal-Code und in der vierten Spalte die Umsetzung im Smart Card-Code beschrieben. Im Anschluss an die Tabelle werden die einzelnen Punkte erläutert.

	UML (und MEL)	Terminal-Code	Smart Card-Code
1.	primitive Datentypen: Number, Boolean, String	Java-Typen: int, bool, String	Java Card-Typen: short, bool, Byte-Array
2.	Sicherheitsdatentypen und MEL-Operationen	Java-Klassen mit statischen Methoden	Java-Klassen mit statischen Methoden
3.	Zertifikatklasse und MEL-Operationen	Java-Klasse mit (statischen) Methoden	Java-Klasse mit (statischen) Methoden
4.	Stereotypen ≪PlainData≫, ≪SignData≫ und ≪HashData≫	Interfaces	Interfaces
5.	Daten- und Nachrichtenklasse	Java-Klasse	Java-Klasse
6.	Komponentenklasse	Java-Klasse	Java-Klasse (extends Applet)
7.	Zustandsklasse einer Komponente	Klasse mit Konstanten	Klasse mit Konstanten
8.	Assoziation mit Multiplizität größer als Eins und Listenoperationen in MEL	Listenklasse mit Zugriffsmethoden	Listenklasse mit Zugriffsmethoden
9.	Klasse mit Stereotyp ≪Constants≫	Klasse mit Konstanten	Klasse mit Konstanten
10.	manuelle Methode (mit Stereotyp ≪Manual≫)	Methodensignatur	Methodensignatur
11.	Stereotyp ≪Initialize≫ für die Attribute und Assoziationsenden einer Komponentenklasse	Konstruktor für die Terminalklasse	Methode für die Initialisierung

Tabelle 6.1.: Im UML-Modell verwendete Elemente und ihre Abbildung auf Quellcode

Im Folgenden wird die Tabelle erläutert:

1. Der in UML definierte Typ `Number` wird im Terminal-Code in den primitiven Typ `int` und im Smart Card-Code in den primitiven Typ `short` übersetzt. Sendet ein Terminal einen Integer an eine Smart Card, wird dieser dort (während der Deserialisierung) als Short-Wert gespeichert und weiterverarbeitet. Ist der Integer als Short-Wert nicht darstellbar, d.h. würde es bei Speicherung zu einem Over- oder Underflow kommen, wird bei der Deserialisierung eine Exception geworfen (siehe Anschnitt 6.2.4). Der in UML definierte Typ Boolean wird im generierten Code in den primitiven Java-Typ `bool` übersetzt. Der in UML definierte Typ String wird im Terminal-Code in den Typ String und im Smart Card Code in ein Byte-Array übersetzt. Dieses speichert die ASCII-Repräsentation des Strings.

2. Die in SecureMDD definierten Sicherheitsdatentypen werden in Java-Klassen übersetzt. Die in MEL vordefinierten kryptographische Operationen sind als Methoden dieser Klassen realisiert. Im Detail wurde die Abbildung in Code in Abschnitt 6.2.2 erläutert. Die vollständige Implementierung ist in Anhang C.1 angegeben.

   Ist ein Assoziationsende im UML-Modell mit dem Stereotyp ≪encrypted≫, ≪encryptedAsymm≫, ≪hashed≫, ≪signed≫ oder ≪MAC≫ annotiert, bekommt das entsprechende Feld in der Java-Klasse den zugehörigen Java-Sicherheitsdatentyp als Typ. Dies sind für den Stereotyp ≪encryptedSymm≫ der Typ `EncDataSymm`, für ≪encryptedAsymm≫ der Typ `EncryptedAsymm`, für ≪hashed≫ der Typ `HashedData`, für den Stereotyp ≪signed≫ der Typ `SignedData` und für ≪MAC≫ der Typ `MACData`. Diese Abbildung ist nachfolgend anhand eines signierten Datums erläutert.

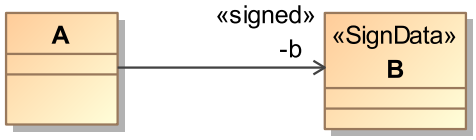

Abbildung 6.7.: Klasse `A` speichert die Signatur, gebildet über einer Instanz von `B`

Die in Abbildung 6.7 dargestellte Klasse `A` mit der Assoziation auf die Klasse `B`, deren Ende mit ≪signed≫ annotiert ist, wird in folgenden Java-Code übersetzt:

```
public class A {
SignedData b; }
```

3. Für jede in UML mit dem Stereotyp ≪Certificate≫ annotierte Klasse wird im Code eine Java-Klasse generiert. Die in MEL vordefinierte Methode `generateCertificate` wird in eine statische Methode dieser Klasse übersetzt. Die ebenfalls vordefinierte Methode `verifyCertificate` ist eine Instanzenmethode dieser Klasse. Die genaue Abbildung der modellierten Zertifikatklasse in Java-Code ist in Abschnitt 6.2.2 erläutert.
4. Für die Stereotypen ≪PlainData≫, ≪SignData≫ und ≪HashData≫ wird jeweils ein gleichnamiges Interface erzeugt. Alle Klassen, die im UML-Modell mit einem dieser Stereotypen annotiert sind, implementieren im generierten Code das entsprechende Interface.
5. Jede Daten- und Nachrichtenklasse sowie die Benutzernachrichtenklassen (im Terminal-Code) werden jeweils in eine Java-Klasse übersetzt. Für die abstrakten Klassen mit Stereotyp ≪Message≫ bzw. ≪Usermessage≫ wird ebenfalls eine Java-Klasse erzeugt, von der die (Benutzer-)Nachrichtenklassen abgeleitet sind. Für jedes Attribut und Assoziationsende der Daten- oder (Benutzer-) Nachrichtenklasse wird im Smart Card-Code in der Java-Klasse ein Feld mit Sichtbarkeit `public` generiert. Im Terminal haben die Felder die Sichtbarkeit `private` und es werden entsprechende `get`- und `set`-Methoden generiert. Dies erleichtert das Testen der Anwendung. Jede Daten- und (Benutzer-) Nachrichtenklasse in Java besitzt einen Konstruktor ohne Argumente, der alle Felder mit einem vordefinierten (Objekt-)Wert initialisiert. Zusätzlich wird ein Konstruktor

generiert, der Parameter für alle Felder der Klasse besitzt und diese entsprechend der übergebenen Argumente initialisiert. Für jede Daten- und (Benutzer-)Nachrichtenklasse wird außerdem eine `copy`-Methode generiert, die als Argument ein Objekt vom selben Typ wie die Klasse bekommt und die Werte der Felder dieses übergebenen Objekts (rekursiv) in die Felder der Klasse kopiert. Diese wird für die Realisierung der (auch in MEL und dem formalen Modell vorhandenen) Kopiersemantik auf Codeebene benötigt. Die Kopiersemantik wird genauer in Abschnitt 10.3.4 beschrieben. Für jede Daten- und (Benutzer-)Nachrichtenklasse wird außerdem eine `equals`-Methode generiert. Diese bekommt als Argument ein Objekt vom gleichen Typ übergeben und überprüft rekursiv, ob die Felder des übergebenen Objekts und die Felder des Objekts, auf dem die Methode aufgerufen wurde, gleich sind. Gleichheit bedeutet in diesem Kontext, wie auch im gesamten SecureMDD-Ansatz, dass die Felder die gleichen Werte haben (tiefe Gleichheit).

Listing 6.23 zeigt beispielhaft den Smart Card-Code der Nachrichtenklasse `Load`.

```
public class Load extends Message {
public short amount;
public HashedData authterminal;

public Load() {
   amount = 0;
   authterminal = new HashedData(); }

public Load(short amount, HashedData authterminal) {
   this.amount = amount;
   this.authterminal = authterminal; }

public byte getCode() {
   return Code.LOAD; }

public boolean equals(Load other) {
   if (this.amount != other.amount)
      return false;
   if (!this.authterminal.equals(other.authterminal))
      return false;
   return true; }

public void copy(Load from) {
   this.amount = from.amount;
   this.authterminal.copy(from.authterminal); }}
```

Listing 6.23: Generierter Code der Klasse `Load`

6. Eine Komponentenklasse wird in eine Java-Klasse übersetzt. Eine Smart Card-Komponentenklasse ist von der Klasse `SimpleComm` abgeleitet, die die Kommunikation kapselt und von der Klasse `javacard.framework.Applet` abgeleitet ist. Abstrakte Komponentenoberklassen (wie die UML-Klasse `Terminal` in der Kopierkartenanwendung) werden bei der Generierung des Codes entfernt. Jede konkrete Subklasse erhält

dann alle Attribute und Assoziationen der Superklasse. Die Definition von gemeinsamen Oberklassen für Komponenten erleichtert die Modellierung von Protokollen, die auf mehreren Komponententypen ausgeführt werden können, zum Beispiel das Abfragen des Kontostands (am Kopier- und dem Ladeterminal). Die Komponentenklassen bzw. ihre Instanzen werden jedoch auf unterschiedlichen Karten bzw. Terminals deployed, weswegen eine gemeinsame Oberklasse im generierten Code keinen Sinn mehr ergibt. Die Attribute und Assoziationsenden einer Komponentenklasse werden im Code in entsprechende Felder mit gleichem Namen übersetzt. Die Felder haben auf der Smart Card die Sichtbarkeit `public`. Wie auch bei den Daten- und Nachrichtenklassen haben die Felder der Terminals die Sichtbarkeit `private` und sind über generierte `get-` und `set`-Methoden zugreifbar. Für jede Komponentenklasse wird ein Konstruktor ohne Argumente erzeugt, der die Felder mit einen vordefinierten (Objekt-)Wert initialisiert. Für eine Terminalklasse wird zusätzlich ein parametrisierter Konstruktor erzeugt, der einen Parameter für jedes Attribut der Klasse besitzt, das mit dem Stereotyp ≪Initialize≫ annotiert ist. Dieser ermöglicht das einfache Initialisieren der Felder zum Testen der Anwendung.

Listing 6.24 zeigt die Implementierung der Smart Card-Klasse `Copycard`. Die `process`-Methoden, die die eigentlichen Protokollschritte implementieren, sowie die Methoden zur Initialisierung der Kopierkarte werden später in diesem Abschnitt vorgestellt und sind hier nicht abgebildet.

```
public class Copycard extends SimpleComm {
   public short balance;
   public Secret passphrase;
   public Nonce challenge;
   public byte statecard;

   public Copycard() {
      balance = 0;
      passphrase = new Secret();
      challenge = new Nonce();
      statecard = StateCard.APPLET_UNINITIALIZED;}
..}
```

Listing 6.24: Generierter Code der Klasse `Copycard`

7. Da Java Card (2.2.1) keine Enumeration-Klassen unterstützt, wird eine UML-Enumeration, die die möglichen Zustände einer Komponentenklasse beschreibt, auf eine Java-Klasse abgebildet. Diese enthält eine Konstante vom Typ `byte` für jedes Enumeration-Literal. Zusätzlich gibt es in der Zustandsklasse einer Kartenkomponente eine Konstante namens `APPLET_UNITIALIZED`, in dem sich die Karte initial, d.h. nach dem Erzeugen des Applets, befindet. Die Werte der Konstanten sind von eins an aufsteigend durchnummeriert.

   Die UML-Assoziation einer Komponentenklasse zu einer Zustandklasse (dessen Assoziationsende mit dem Stereotyp ≪status≫ annotiert ist, wird in ein Feld vom Typ `byte` in der Komponentenklasse übersetzt (siehe Feld `statecard` in Listing 6.24).

8. Im UML-Modell sind Listen durch eine Assoziation mit Multiplizität größer als eins modelliert. Der Zugriff auf diese Liste ist nur durch den Aufruf der vordefinierten MEL-Listen-Operationen möglich.

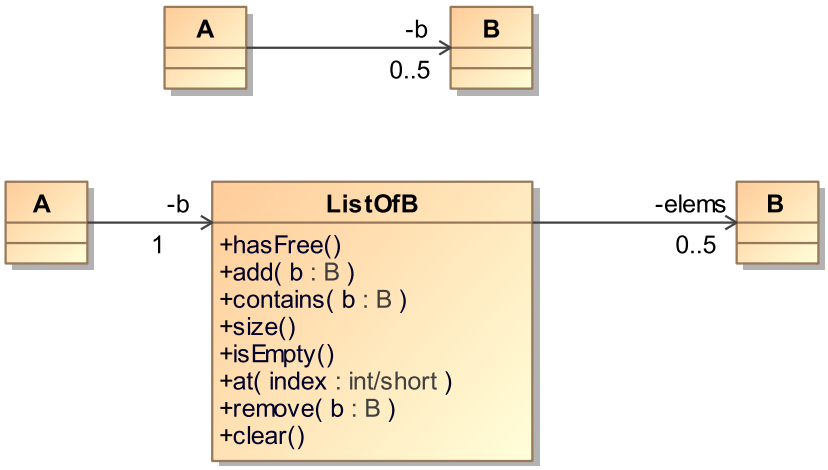

Abbildung 6.8.: Eine Liste in UML (oben) und im Java-Code (unten)

Abbildung 6.8 zeigt, wie eine in UML modellierte Liste (Abb. oben) im Java-Code abgebildet wird (Abb. unten). Für die Klasse `B`, auf die die Assoziation zeigt, wird eine Containerklasse mit dem Namen `ListOfB` generiert, die die Listenelemente vom Typ `B` verwaltet. Die Klasse `A` referenziert nur noch diese Containerklasse, die die Methoden für den Zugriff auf die Liste bereitstellt. Dies sind die in MEL definierten Listenoperationen. Die eigentliche Liste ist in der Klasse `ListOfB` durch ein Array implementiert. In dem in der Abbildung dargestellten Beispiel hat dieses die Länge fünf. Aufgrund des begrenzten Speicherplatzes auf der Karte sind dort keine Listen unbegrenzter Länge (d.h. Assoziationen mit Multiplizität `*` oder `n`) erlaubt. Auf Terminalseite ist ihre Verwendung jedoch möglich. Sie sind durch eine Java-`ArrayList` implementiert. Die Implementierung der Klasse `ListOfB` ist im Anhang C.2 abgebildet.

9. Eine mit dem Stereotyp ≪Constants≫ annotierte UML-Klasse definiert Konstanten. Die Modellierung wurde in Abschnitt 4.2.1, Seite 60 beschrieben. Eine solche Klasse wird auf eine Java-Klasse abgebildet. Für jedes Attribut der UML-Klasse wird eine Konstante generiert. Ist für ein Attribut kein Typ angegeben, wird der Typ `short` (im Smart Card-Code) bzw. `int` (im Terminal-Code) verwendet. Ist für Attribute vom Typ `Boolean` kein Wert angegeben, wird ihnen der Wert `false` zugewiesen. Ist für Attribute vom Typ `Number` (die bei der Generierung in Felder vom Typ `short` bzw. `int` übersetzt werden) kein Wert angegeben, wird ihnen ein Wert zugewiesen. Hierfür werden Werte von eins an aufsteigend verwendet.

10. Die Deklaration einer manuellen Methode, d.h. einer UML-Operation, die mit dem Stereotyp ≪Manual≫ annotiert ist, wird im Code in die Signatur dieser Methode übersetzt. Der Rumpf muss nach der Generierung des Codes von Hand implementiert werden. Die Transformationen berücksichtigen bei einer Neugenerierung des Codes die schon vorhandene Implementierung und der von Hand erstellte Code wird in diesem Fall nicht

überschrieben bzw. gelöscht. Ist eine UML-Klasse mit dem Stereotyp ≪Manual≫ annotiert, wird diese in eine Java-Klasse übersetzt. Für jede ihrer UML-Operationen wird ebenfalls eine Methodensignatur generiert.

11. Im UML-Modell gibt der Stereotyp ≪Initialize≫ an, welche Attribute und Assoziationsenden der Komponentenklassen initialisiert werden müssen. Die eigentlichen Werte, mit denen die Komponenten initialisiert werden sollen, sind jedoch nicht im UML-Modell angegeben. Stattdessen werden die konkreten Initialisierungsdaten erst beim Deployment der Anwendung festgelegt. Der Grund hierfür ist, dass diese Daten bzw. für die Initialisierung geltende Einschränkungen zum Teil sehr schwer zu formalisieren sind. Ein Beispiel ist das Bezahlsystem Mondex, bei dem eine Voraussetzung ist, dass alle ausgegebenen Bezahlkarten unterschiedliche Namen besitzen.

    Während die Initialdaten eines Terminals beim Erzeugen des Terminalobjekts übergeben werden, ist die Initialisierung auf Smart Card-Seite etwas komplizierter. Da das Applet von der Java Card-Laufzeitumgebung (durch Aufruf der `install()`-Methode) erzeugt wird, muss die Initialisierung separat stattfinden. Das Applet befindet sich deshalb vor der Initialisierung in einem Zustand namens `APPLET_UNINITIALIZED`, der beim Erzeugen des Applets gesetzt wird. Dieser Zustand bedeutet, dass die Smart Card zunächst eine APDU erwartet, die die Initialisierung der Karte vornimmt. Bevor diese gesendet wird, reagiert die Karte auf jede andere empfangene Nachricht mit dem Abbruch des Protokollschrittes und dem Werfen einer Exception. Die Nachricht, die die Initialdaten enthält, bewirkt, dass die Methode `initialize_applet()` auf der Karte aufgerufen wird. Diese hat einen Parameter für jedes Attribut und Assoziationsende der Karte, das initialisiert werden muss, und initialisiert die Felder der Karte entsprechend. Unter anderem wird hier auch der Zustand der Karte in den Anfangszustand gesetzt (üblicherweise IDLE). Anschließend kann die Karte weitere, im UML-Modell modellierte, Nachrichten empfangen.

    Der generierte Terminal-Code enthält eine Methode für jede Kartenkomponentenklasse, mit der diese Karte initialisiert werden kann. Diese Methode erstellt mithilfe der übergebenen Initialdaten die APDU, die die Karte initialisiert.

6.3.2. Transformation der Aktivitätsdiagramme in Code

Ein Protokoll (wie es in einem Aktivitätsdiagramm modelliert ist) besteht aus mehreren Protokollschritten. Am Ende eines Protokollschritts wird eine Nachricht an eine andere Komponentenklasse gesendet, die bewirkt, dass der nächste Protokollschritt startet. Bei der Abbildung eines Aktivitätsdiagramms müssen somit die Übersetzung der Kommunikation in Code, d.h. der Versand von Nachrichten zwischen Terminal und Karte, sowie die Transformation der eigentlichen Protokollschritte betrachtet werden.

Tabelle 6.2 gibt einen Überblick über die Übersetzungsregeln der UML-Aktivitätsdiagramme (inklusive der MEL-Ausdrücke) in Code. Die einzelnen Punkte sind im Anschluss an die Tabelle ausführlich beschrieben.

	UML und MEL	Terminal-Code	Smart Card-Code
1.	Kommunikation zwischen Terminal und Smart Card	Versand bzw. Empfang eines Byte-Arrays	Versand bzw. Empfang eines Byte-Arrays
2.	Protokollschritt im Aktivitätsdiagramm	Java-Methode	Java-Methode
3.	Kopiersemantik MEL	Referenzsemantik	Referenzsemantik
4.	Erzeugen eines neuen Objekts mit `create`	Erzeugen eines neuen Objekts mit `new`	Herausgabe eines Objekts durch den Objektmanager
5.	Lokale Variablen in MEL	Lokale Variablen	Lokale Variablen
6.	Aufruf eines Subdiagramms	Aufruf einer Methode	Aufruf einer Methode
7.	Arithmetische Operationen (in MEL)	Java-Methoden	Java-Methoden mit Over-/Underflow-Test
8.	„== “-Operator zum Test auf Gleichheit zweier Objekte	Aufruf der `equals`-Methode	Aufruf der `equals`-Methode
9.	Abbruch eines Protokollschritts (durch Erreichen eines UML-`FlowFinal` Knotens oder implizit durch Werfen einer Exception in einer MEL-Operation)	Abbruch des Protokollschritts durch Werfen einer Exception	Abbruch des Protokollschritts durch Werfen einer Exception und Senden eines Fehlercodes an das Terminal

Tabelle 6.2.: Abbildung der Aktivitätsdiagramme in Code

1. Die Abbildung der nachrichtenbasierten Kommunikation im UML-Modell auf die Byte-Array basierte Kommunikation auf Codeebene wurde bereits in den Abschnitten 6.2.3 und 6.2.4 erläutert.
2. Der Empfang einer Nachricht führt auf der Smart Card sowie im Terminal zum Aufruf einer Methode, die den zugehörigen Protokollschritt durchführt. Diese ist in der entsprechenden Komponentenklasse implementiert und beinhaltet den Namen der Nachricht, mit dessen Empfang der Protokollschritt beginnt (diese Methode wurde in Abschnitt 6.2.3 beispielhaft als `processMsgName` bezeichnet). Zum Beispiel realisiert die Methode `processLoad` der Klasse `Copycard` den Protokollschritt, der im UML-Modell auf den Empfang der `Load`-Nachricht folgt. Der entsprechende Ausschnitt aus dem UML-Aktivitätsdiagramm ist in Abbildung 6.9 dargestellt. Die gesamten Aktivitätsdiagramme der Anwendung sind in Abschnitt 4.5.2 abgebildet.

   In Listing 6.25 ist die Methode `processLoad` dargestellt, die diesen Protokollschritt implementiert. Sie wird von der Methode `process(Message msg)` aufgerufen, die

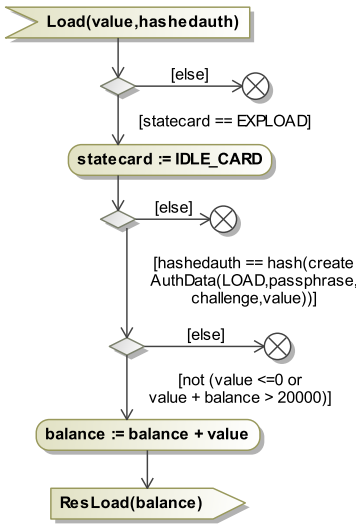

Abbildung 6.9.: Protokollschritt `Load`

anhand des Typflags der Klasse (der über die Methode `getCode()` abgefragt werden kann) erkennt, dass die empfangene Nachricht den Typ `Load` hat.

```
1  public void processLoad(Load inmsg) {
2     short value = inmsg.amount;
3     HashedData hashedauth = inmsg.authterminal;
4
5     if (statecard == StateCard.EXPLOAD) {
6        statecard = StateCard.IDLE_CARD;
7        AuthData ad = Store.newAuthData(Constants.LOAD,
8              passphrase, challenge, value);
9        if (((hashedauth.equals(HashedData.hash(ad)) &&
10             (value > (short) 0)) &&
11             (Math.plus(value, balance) <= (short) 20000))){
12          balance = Math.plus(balance, value);
13          sendMsg(Store.newResLoad(balance));
14       } else {
15          stop();  }
16    }
17    else {
18       stop();}
19 }
```

Listing 6.25: Methode `processLoad` der Klasse `Copycard`

Da die Sprache MEL ähnlich zu der Sprache Java ist, ist die Übersetzung der MEL-Ausdrücke in Java Card- bzw. Java-Code an vielen Stellen trivial. Dennoch gibt es einige Besonderheiten bei der Übersetzung der mit UML und MEL modellierten Protokollschritte, auf die in den folgenden Punkten eingegangen wird.

3. MEL ist wertbasiert, Referenzen auf Objekte sowie Objektidentitäten spielen eine untergeordnete Rolle. MEL verwendet somit Wertgleichheit, d.h. zwei Objekte sind gleich, wenn ihre Attribute und Assoziationsenden die gleichen Werte haben. Java dagegen besitzt eine Referenzsemantik. Für die Korrektheit des generierten Codes, d.h. die Verfeinerungsbeziehung zu dem generierten formalen Modell, ist es jedoch essentiell, dass der generierte Code ebenfalls wertbasiert ist und eine Kopiersemantik besitzt. Die entsprechende Umsetzung im Code ist ausführlich in Abschnitt 10.3.4 erläutert.
4. In MEL werden Objekte durch die Verwendung des Schlüsselwortes `create` erzeugt. Im generierten Terminal-Code entspricht dies dem Java-Schlüsselwort `new`. Auf der Smart Card werden die Objekte von dem Objektmanager (Klasse `Store`) zur Verfügung gestellt. Das Schlüsselwort `create` wird deshalb im Smart Card-Code in den Aufruf der entsprechenden `new`-Methode der Klasse `Store` übersetzt (siehe Zeilen 7-8 des Listings 6.25). Der Objektmanager ist in Abschnitt 6.2.5 ausführlich erläutert.
5. MEL erlaubt die Deklaration von lokalen Variablen. Diese werden auch im generierten Java-Code in entsprechende Variablendeklarationen abgebildet. Eine Besonderheit ist die Deklaration von lokalen Variablen für die Parameter der empfangenen Nachricht zu Beginn eines Protokollschritts. Dies ist in Listing 6.25 (Zeilen 2-3) für das Beispiel der `Load`-Nachricht zu sehen. Die lokalen Variablen `value` und `hashedAuth` speichern die in den Feldern `amount` und `authhashed` der `Load`-Nachricht enthaltenen Werte. Die Namen der Variablen ergeben sich aus dem Aktivitätsdiagramm.
6. Für jedes in UML modellierte Subdiagramm wird eine eigene Methode generiert. Der Name der Methode entspricht dem Namen des Subdiagramms, ihre Ein- und Ausgabeparameter ergeben sich aus den im Subdiagramm modellierten UML-`ActivityParameterNodes`. Der Aufruf eines UML-Subdiagramms wird in den Aufruf der gleichnamigen Methode übersetzt.
7. Eine weitere Besonderheit ist die Übersetzung der in MEL verwendeten arithmetischen Operatoren (z.B. + und −) in Code.

```
public static short plus(short x, short y) {
   short res = (short) (x + y);
   if (x < 0 && y < 0 && res >= 0)
      ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   if (x > 0 && y > 0 && res < 0)
      ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   return res; }
```

Listing 6.26: Methode `plus` der Klasse `Math`

Die Operatoren werden auf Terminalseite in einen Aufruf der entsprechenden Java-Operatoren übersetzt. Während im Terminalcode diese Operatoren auf Integer aufgerufen werden (da der Typ `Number` in Integer übersetzt wird), werden die Operatoren im Smart Card-Code auf Short-Werte angewendet. Da der Wertebereich des primitiven Typs `short` relativ klein ist (−32.768 bis 32.767), wird vor Ausführung der Operation überprüft, ob das Ergebnis außerhalb des gültigen Wertebereichs liegt und somit ein Over- bzw. Underflow auftreten würde. Die Annahme ist, dass dieser Over- bzw. Underflow von dem Modellierer nicht gewollt ist und somit einen Fehler darstellt. Die

arithmetischen Operationen sind aus diesem Grund auf der Smart Card in Methoden ausgelagert, die in der Klasse `Math` definiert sind. Zum Beispiel gibt es für die Addition zweier Short-Werte die Methode `plus`. Ihre Implementierung ist in Listing 6.26 dargestellt.

Die Methode prüft, ob während der Addition der beiden Argumente `x` und `y` ein Over- bzw. Underflow aufgetreten ist. In diesem Fall wirft die Methode eine ISOException. Der aktuell durchgeführte Protokollschritt wird somit an dieser Stelle abgebrochen. Tritt kein Over-/Underflow auf, wird das Ergebnis `res` der Addition zurückgegeben. Ein Beispiel für den Aufruf der Methode `plus` zeigt Listing 6.25 (Zeile 12).

8. Der in MEL verwendete ==-Operator prüft die Wertgleichheit zweier Objekte. Der ==-Operator in Java testet jedoch die Objektgleichheit. Da der generierte Code jedoch eine Kopiersemantik besitzt, wird für jede Klasse eine `equals`-Methode benötigt, die rekursiv die Werte der Felder der Objekte miteinander vergleicht. Diese Methode wird für jede Java-Klasse automatisch generiert. Der MEL-Operator == wird somit bei der Generierung des Codes in einen Aufruf der Methode `equals` übersetzt. Ein Beispiel hierfür zeigt Listing 6.25 (Zeile 9), in der zwei Objekte vom Typ `HashedData` miteinander verglichen werden.

9. Im UML-Modell modelliert ein UML-`FlowFinal` Knoten das Auftreten eines Fehlers, d.h. das Werfen einer Exception. Zusätzlich können auch in MEL vordefinierte Methoden, zum Beispiel das Hinzufügen eines Elements in eine volle Liste (in der Methode `add`), eine Excpeption werfen. Im generierten Code wird ein UML-`FlowFinal` Knoten in den Aufruf der Methode `stop()` übersetzt (siehe z.B. Listing 6.25, Zeile 15). Diese wird ebenfalls aufgerufen, wenn in einer vordefinierten Methode ein Fehler auftritt. Der Smart Card-Code dieser Methode ist in Listing 6.27 dargestellt.

```
public static void stop() {
   ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED); }
```

Listing 6.27: Methode `stop` der Klasse `SimpleComm`

Die Methode beendet den aktuellen Protokollschritt auf der Karte durch Werfen einer ISOException. Dies bewirkt, dass der Fehlercode `SW_CONDITIONS_NOT_SATISFIED` an das Terminal zurückgesendet wird. Aus Sicherheitsgründen wird dabei für jede Art von Fehler derselbe Fehlercode verwendet. Auf diese Weise erlangt ein Angreifer kein Wissen darüber, welcher Fehler auf der Karte aufgetreten ist. Das Terminal beendet nach dem Empfang des Fehlercodes den aktuellen Protokollschritt und somit den gesamten Protokolllauf. Auf Terminalseite wirft die `stop()`-Methode eine `TerminalException`. Auch hier wird für jede Art von Fehler dieselbe Exception geworfen. Das Werfen einer `TerminalException` führt auf Terminalseite ebenfalls zum Abbruch des Protokollschritts und beendet somit den Protokolllauf.

6.3.3. Transformation der Deploymentdiagramme in Code

Das Deploymentdiagramm wird bei der Generierung des Quellcodes für die Berechnung der Portnamen verwendet (siehe Abschnitt 6.2.3). Anders als für die Generierung der formalen

Spezifikation, bei der zusätzlich die Angreiferfähigkeiten berücksichtigt werden müssen, hat das Deploymentdiagramm bei der Codegenerierung somit eine eher geringe Bedeutung.

In diesem Kapitel wurde die Generierung des lauffähigen Quellcodes aus dem plattformunabhängigen UML-Modell einer Anwendung erläutert. Im folgenden Abschnitt wird dieser Teil der Arbeit in den Kontext anderer Forschungsarbeiten gesetzt, die sich ebenfalls mit der Generierung von Quellcode befassen.

6.4. Verwandte Arbeiten

Die für diese Dissertation wichtigen Arbeiten anderer Forschergruppen zum Thema Codegenerierung lassen sich in zwei Gruppen unterteilen. Dies sind Ansätze, die Smart Card-Code generieren sowie Ansätze, die aus UML-Aktivitätsdiagrammen Code generieren. Natürlich gibt es darüber hinaus weitere Arbeiten zur Codegenerierung aus Modellen allgemein, z.B. die Generierung von Codefragmenten (wie Klassen und Methodensignaturen) aus UML-Modellen wie sie auch mittlerweile von vielen Modellierungstools unterstützt wird. Da diese Arbeiten aber keinen Bezug zur Entwicklung von sicherheitskritischen Anwendungen haben, werden sie in dieser Arbeit nicht diskutiert.

Generierung von Smart Card-Code
Coglio und Green [37–39] stellen ein Werkzeug zur Verfügung, mit dem korrekter Java Card-Code generiert werden kann. Ein Applet wird in der textuellen Sprache *SmartSlang* spezifiziert. Das Ziel der Sprache ist es, eine Abstraktion zur Programmierung mit Java Card zu erreichen, die Sprache ist jedoch relativ eng an die Sprache Java Card angelehnt. Die erstellte Spezifikation wird mithilfe eines Codegenerierungtools, das einen Theorembeweiser integriert, in Java Card übersetzt. Der generierte Code ist eine Verfeinerung der Spezifikation. Damit überprüfbar ist, dass der durchgeführte Verfeinerungsbeweis korrekt ist, werden die gemachten Beweise exportiert und können von einem separaten Beweischecker überprüft werden. Coglio und Green betrachten nur die funktionale Korrektheit eines einzelnen Applets. Der Code für die Terminals kann mit dem Ansatz nicht entwickelt werden. Auch die Sicherheit der entwickelten Anwendung (bzw. des Applets) wird in dem Ansatz nicht betrachtet. Dies sind Unterschiede zu dem in dieser Arbeit vorgestellten Ansatz. Ein weiterer Vorteil des SecureMDD-Ansatzes ist die starke Abstraktion des plattformunabhängigen UML-Modells von der Codeebene. Hier muss der Modellierer keine Kenntnisse über die Programmierung in Java Card besitzen. Dies ist bei der Erstellung einer *SmartSlang*-Spezifikation nicht der Fall.

Jan Jürjens et al. verfolgen verschiedene Ansätze, um die Implementierung von kryptographischen Protokollen als sicher nachweisen zu können. In [115] evaluieren Lloyd und Jürjens, inwiefern die Verwendung des UMLSec-Tools in Kombination mit der JML (Java Modeling Language [32]) die Korrektheit und Sicherheit einer Implementierung garantieren kann. Dies geschieht anhand einer Industriefallstudie, ein auf Smart Cards basierendes biometrisches Authentisierungssystem. Ziel war es, Teile des von Hand geschriebenen Codes gegen die UMLSec-Spezifikation der Anwendung zu verifzieren. Viele der durchzuführenden Checks konnten jedoch mit JML nicht adäquat spezifiziert werden. Ein Unterschied zu dem in dieser Arbeit vorgestelltem Ansatz ist, dass das Erstellen des Codes ein manueller Schritt ist und die Implementierung außerdem nur teilweise überprüft wurde. Der SecureMDD-Ansatz dagegen generiert automatisch den Quellcode einer Anwendung. Die Transformationen sind auf eine

Weise implementiert, dass der generierte Code eine Verfeinerung des abstrakten Modells ist und die Sicherheit auch für die Codeebene gilt.

In [96] stellt Jürjens einen zu [115] ähnlichen Ansatz vor, mit dem ein annotiertes Java (Card)-Programm mithilfe von automatischen Theorembeweisern verifiziert werden kann. Der Ansatz wurde unter anderem am Smart Card-Beispiel „Common Electronic Purse Specification“ (eine Spezifikation für eine elektronische Geldbörse) evaluiert. Der handgeschriebene annotierte Quellcode wird in ein abstraktes Modell der Anwendung transformiert, das von den angeschlossenen automatischen Theorembeweisern auf Einhaltung der annotierten Sicherheitseigenschaften überprüft wird.

Nikseresht et al. [145] und Mostowski [140] generieren ebenfalls Smart Card-Code aus UML-Modellen. Beide Ansätze wurden bereits in den Abschnitten 3.4 und 4.6 diskutiert.

Das BOM-Projekt (B With Optimised Memory) [182] beschäftigt sich mit der automatischen Generierung von Java Card-Code aus B Spezifikationen. Eine abstrakte B-Spezifikation wird schrittweise in immer konkretere Spezifikationen verfeinert. Der letzte Generierungsschritt erzeugt aus einer konkreten Spezifikation Java Card-Code. Für diesen ist es jedoch nicht möglich zu zeigen, dass der Code eine Verfeinerung der konkreten Spezifikation ist. Anders als im SecureMDD-Ansatz muss der Verfeinerungsbeweis für jede betrachtete Anwendung durchgeführt werden.

Spi2Java [156] ist ein Tool, mit dem eine Spezifikation eines kryptographischen Protokolls im Spi-Kalkül [1] erstellt wird. Auf diesem Modell können dann Sicherheitseigenschaften bewiesen werden. Anschließend wird die Spezifikation mithilfe eines Codegenerators in Java-Code übersetzt. Die Korrektheit der Codegenerierung oder eine formale Verfeinerung wird jedoch nicht bewiesen. Damit lassen sich ebenfalls keine garantierten Aussagen über die Sicherheit und Korrektheit der Implementierung aussagen.

Generierung von Quellcode aus Aktivitätsdiagrammen
Verschiedene Arbeiten beschäftigen sich mit der Generierung von (Java-)Code aus UML-Aktivitätsdiagrammen. In dem Projekt MDD4SOA von Mayer et al. [120] werden UML-Aktivitätsdiagramme für die Modellierung von Service-Orchestrierung verwendet und unter anderem in Java-Code übersetzt. Hierfür wird die UML um das Profil UML4SOA erweitert, eine eigene Sprache, vergleichbar mit MEL, wird jedoch nicht definiert.

Kraus et al. [110] verwenden Aktivitätsdiagramme, um Businessprozesse und ihren Kontrollfluss für webbasierte Anwendung zu modellieren und anschließend aus den Modellen Quellcode zu generieren. Die Laufzeitumgebung des Tools UWE (UML-based Web Engineering) unterstützt dann die Ausführung der Aktivitäten. Die Inhalte der verwendeten UML-Actions werden jedoch, anders als in SecureMDD, nicht interpretiert.

[21], [62] und [186] beschäftigen sich mit der Generierung von Code aus Aktivitätsdiagrammen allgemein, d.h. unabhängig von einer speziellen Anwendungsdomäne.

Keine der genannten Arbeiten verwenden Aktivitätsdiagramme jedoch auf die Art wie es in dieser Arbeit getan wird. Die Kombination von Aktivitätsdiagrammen und Ausdrücken der Sprache MEL ermöglicht im SecureMDD-Ansatz die automatische Generierung einer vollständigen lauffähigen Implementierung. Gleichzeitig wird jedoch die Modellierung einer Anwendung auf einem abstrakten Level unterstützt, die keine Kenntnis von Implementierungsdetails verlangt.

7 Modellbasiertes Testen

Zusammenfassung: Ein essentieller Bestandteil des Ansatzes zur Entwicklung einer sicherheitskritischen Anwendung ist die Qualitätssicherung. Ergänzend zu der interaktiven Verifikation von Sicherheitseigenschaften verwendet der SecureMDD-Ansatz modellbasiertes Testen. Dies hat zum Ziel, funktionale Fehler in den Protokollen der zu entwickelnden Anwendung sowie Sicherheitslücken zu einem möglichst frühen Zeitpunkt und auf einfache Weise zu finden. Hierfür werden Testfälle und mögliche Angriffsszenarien mit UML modelliert und aus diesem Modell automatisch Testfälle generiert. Diese werden auf dem generierten Code der Anwendung ausgeführt, um ihn zu testen. Dieses Kapitel beschreibt das modellbasierte Testen in SecureMDD und fasst die Ergebnisse aus der Diplomarbeit von Katkalov [105] zusammen, welche außerdem in [106] publiziert sind.

Abschnitt 7.1 motiviert das modellbasierte Testen einer Anwendung im SecureMDD-Ansatz. Anschließend beschreibt Abschnitt 7.2 die Modellierung von (funktionalen) Testfällen in SecureMDD, Abschnitt 7.3 erläutert das Erstellen und Ausführen von Angriffsszenarien. Zuletzt gibt Abschnitt 7.4 einen kurzen Überblick über die Implementierung der Testunterstützung und setzt diese Arbeit in Abschnitt 7.5 in den Kontext anderer Forschungsarbeiten zu diesem Thema.

7.1. Motivation

Schon während der Entwicklung ist das Testen einer sicherheitskritischen Anwendung eine wichtige Voraussetzung für eine hohe Qualität des Endprodukts. Jedoch sind der Entwurf und das manuelle Erstellen von Testfällen eine mühsame und fehleranfällige Aufgabe. Modellbasiertes Testen ist ein Testverfahren, das vor allem in modellbasierten Ansätzen verwendet wird. Hierbei kann ein an das Anwendungsmodell angelehntes oder darin integriertes Testmodell erstellt und aus diesem Code für die modellierten Testfälle generiert werden. Zum Teil ist es auch möglich, Testfälle direkt aus dem Anwendungsmodell automatisch zu erzeugen. Die benötigten Testdaten können dabei entweder direkt im Testmodell angegeben oder aus einer externen Datenbank ausgelesen werden. Durch die enge Kopplung der Modelle kann eine Änderung des Anwendungsmodells sowie der Testfallmodelle sogar in einem Schritt erfolgen. Ein häufiger Fehler beim Testen einer Anwendung ist, den Entwickler der Software auch als Tester

einzusetzen. Die Entwickler haben zwar das benötigte Wissen über die entwickelte Anwendung, ihnen fehlt aber die für das Testen benötigte Objektivität. So werden einem Entwickler Fehler, die er beim Entwurf gemacht und anschließend implementiert hat, auch beim Testen der Anwendung nicht auffallen. Aus diesem Grund ist es wichtig, dass Entwickler und Tester getrennte Personen sind. Dies gilt insbesondere auch für das modellbasierte Testen. Einen Überblick über modellbasiertes Testen geben [114] und [84].

Die Unterscheidung zwischen funktionalen Tests sowie den Tests der Sicherheit einer Anwendung ist naheliegend. Funktionale Tests überprüfen die Funktionalität der Anwendung. Insbesondere dienen sie dazu, logische Fehler in den kryptographischen Protokollen zu finden. Beispielsweise kann man testen, ob für verschiedene Eingaben fehlerfreie Protokollabläufe möglich oder bestimmte Systemzustände erreichbar sind. Ein Test auf Sicherheitslücken dagegen prüft, ob ein bestimmter Angriff in der modellierten Anwendung möglich ist. Dabei werden konkrete Angriffsszenarien modelliert und überprüft, ob ein Angreifer diese erfolgreich durchführen kann.

Der SecureMDD-Ansatz stellt den Modellierern oder (externen) Testern eine einfache und intuitiv zu verwendende Methode zur Verfügung, um eine Anwendung zu testen, die sich in der Entwicklung befindet. Hierfür wird ein an das Anwendungsmodell angelehntes Testfallmodell in UML erstellt. Die benötigten Testdaten werden dabei ebenfalls direkt im Modell angegeben. Der Ansatz unterstützt sowohl das dynamische Testen der Funktionalität der Anwendung (d.h. ablaufbezogenes Testen) als auch die Simulation eines Angreifers, um mögliche Fehler und Sicherheitslücken auf Codeebene zu erkennen.

Zeitlich gesehen sollte das modellbasierte Testen einer Anwendung der interaktiven Verifikation vorgelagert sein. Der Grund hierfür ist, dass jeder Fehler und jede Sicherheitslücke, die beim Testen gefunden und behoben wird, bei der interaktiven Verifikation Zeit und somit Kosten spart. Findet man einen Fehler bei der Verifikation und korrigiert das UML-Anwendungsmodell entsprechend, müssen die bereits gemachten Beweise wiederholt und eventuell erneuert werden. Das Testen einer Anwendung hingegen ist in wesentlich kürzerer Zeit und mit weniger Aufwand möglich. Der SecureMDD-Ansatz ist auch ohne die Verifikation von Sicherheitseigenschaften verwendbar. Dies bedeutet, dass eine Anwendung mit UML entworfen und anschließend lauffähiger Code generiert werden kann, ohne die Sicherheit der Anwendung zu verifizieren. In diesem Fall ist das modellbasierte Testen der Anwendung essentiell, um Fehler und Sicherheitslücken zu finden.

7.2. Modellierung von Testfällen

Bei der Integration des modellbasierten Testens wurde darauf geachtet, dass die Modellierung der Testfälle intuitiv verständlich und für jemanden, der mit dem SecureMDD-Modellierungsansatz vertraut ist, schnell erlernbar ist. Dem wird unter anderem dadurch Rechnung getragen, dass dieselben Diagrammtypen (insbesondere Sequenz- und Aktivitätsdiagramme) sowie die domänenspezifische Sprache MEL für die Modellierung verwendet werden. Hierfür wurde die Sprache MEL um einige wenige Konstrukte erweitert, die im Detail in [105] beschrieben sind. Um das bestehende Anwendungsmodell durch die Definition der Testfälle nicht zu verändern, wird das Testfallmodell in einem separaten UML-Paket erstellt. Die Testfallmodelle verweisen lediglich auf die Sequenzdiagramme des Anwendungsmodells und verwenden die in den Klassendiagrammen definierten Klassen und Attribute.

7.2.1. Aufbau der Testfalldiagramme

Zunächst wird ein kleines Beispiel für die Modellierung eines funktionalen Testfalls betrachtet. Abbildung 7.1 zeigt einen modellierten Testfall zum Testen der Kopierkartenanwendung. Es wird getestet, dass der Kontostand der Karte beim Abfragen des Kontostands nicht verändert wird und der Kartenbesitzer das richtige Guthaben der Karte mitgeteilt bekommt.

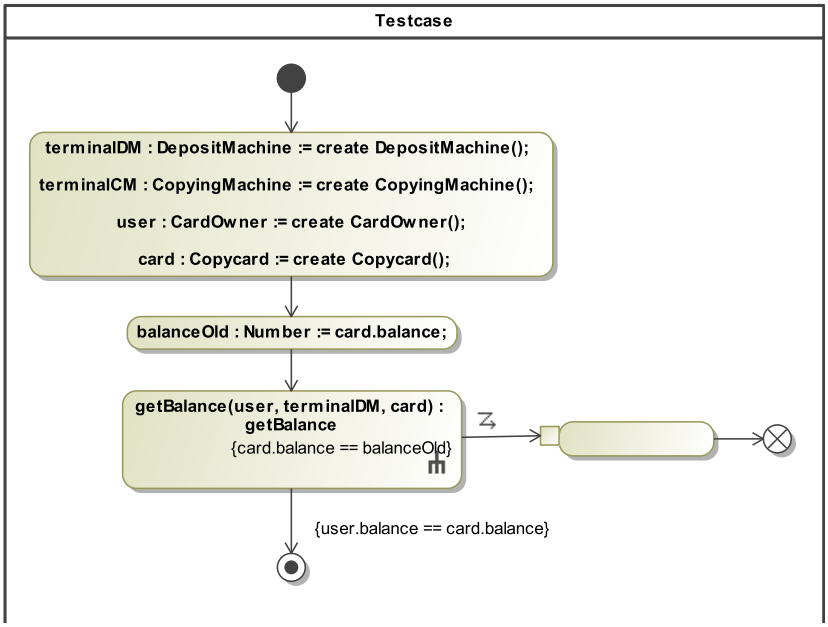

Abbildung 7.1.: Funktionaler Test für das Abfragen des Kontostands einer Kopierkarte

Das Testfalldiagramm initialisiert zunächst die Komponenten der Anwendung. Hierfür wird der parameterlose Konstruktor verwendet. Dies bedeutet, dass alle Attribute und Assoziationen der Komponentenklassen mit Defaultwerten initialisiert werden. Beispielsweise bekommen Attribute vom Typ `Number` initial den Wert 0 und Geheimnisse vom Typ `Secret` alle den gleichen, vorgegebenen Wert. Im Anschluss an die Initialisierung wird eine lokale Variable mit Namen `balanceOld` vom Typ `Number` deklariert und mit dem aktuellen Kontostand der Kopierkarte `card` initialisiert. In diesem Fall ist dies der Wert 0. Dann wird das Protokoll `getBalance` aufgerufen, das den Kontostand der Karte abfragt. Der Aufruf bezieht sich auf das Sequenzdiagramm `getBalance`, das Lebenslinien für die Komponenten `CardOwner`, `DepositMachine` und `Copycard` enthält. Als Argumente werden deshalb die zuvor erzeugten Instanzen dieser Komponentenklassen übergeben. Die Action, die das Protokoll aufruft, enthält zusätzlich den Constraint `card.balance == balanceOld`. Dies ist eine Invariante, die immer, d.h. vor Ausführung des Protokolls, nach jedem Protokollschritt sowie am Ende des Protokolllaufs gelten muss. Wird die Invariante zu einem Zeitpunkt verletzt, schlägt der Test fehl.

Der modellierte Testfall sollte ohne Auftreten eines Fehlers, d.h. dem Werfen einer Exception, ausgeführt werden. In diesem Fall muss der Kontostand der Karte mit dem Wert übereinstimmen, den der `CardOwner` als Antwort erhalten hat. Dies ist als Constraint an den `ControlFlow` annotiert (`user.balance == card.balance`). Die Ausführung des Test-

falls endet an dieser Stelle mit einem UML-`ActivityFinalNode` und der Test ist erfolgreich. Sollte während der Protokollausführung ein Fehler auftreten, soll der Test fehlschlagen. Aus diesem Grund endet der Zweig des UML-`ExceptionHandlers` mit einem UML-`FlowFinalNode`. Wird dieser bei der Ausführung des Testfalls erreicht, schlägt der Test fehl.

Im Folgenden wird der Aufbau eines Testfalldiagramms erläutert. Ein Testfall wird durch ein Aktivitätsdiagramm modelliert und besteht aus mehreren Teilen. Der Aufbau ist beispielhaft in Abbildung 7.2 dargestellt.

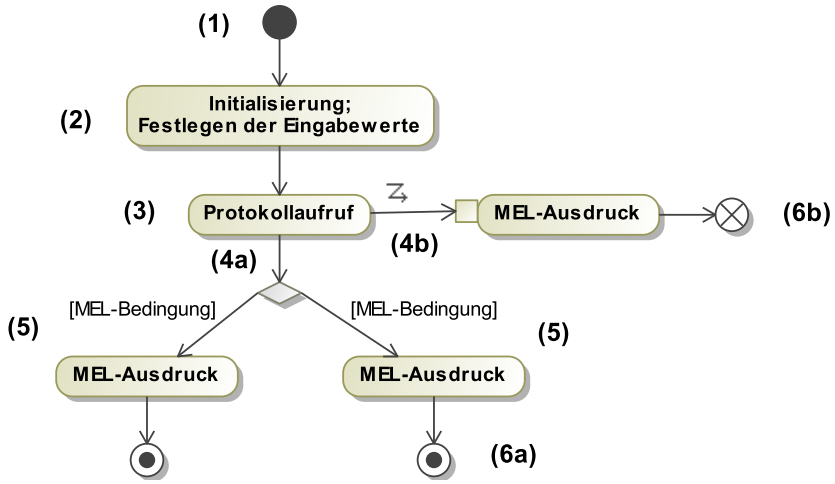

Abbildung 7.2.: Beispielhafter Aufbau eines Testfalldiagramms

1. Jedes Testfalldiagramm beginnt mit einem UML-`InitialNode` als Startknoten.
2. Im Anschluss folgt die Initialisierung der Komponenten sowie das Festlegen der Eingabewerte, mit denen getestet werden soll. Benötigt das zu testende Protokoll keine Benutzereingaben, d.h. hat die Nachricht, die den Protokolllauf startet, keine Argumente, kann dieser Teil weggelassen werden. Die Initialisierung sowie die Definition der Eingabewerte wird mit UML-`Actions` modelliert. Um die Diagramme übersichtlicher zu halten, kann dies auch in einem separaten Aktivitätsdiagramm modelliert werden, das von dem Testfalldiagramm aufgerufen wird.
3. Der nächste Schritt ist der Aufruf des zu testenden Protokolls bzw. des durchzuführenden Angriffs in einer UML-`Action`. Im Falle eines funktionalen Tests wird an dieser Stelle das Sequenzdiagramm, das das zu testende Protokoll enthält, aufgerufen. Soll das Testfalldiagramm testen, ob ein bestimmter Angriff auf die modellierte Anwendung möglich ist, wird an dieser Stelle ein zuvor erstelltes Sequenzdiagramm aufgerufen, das den Angriff beschreibt. Die Modellierung von Angriffsszenarien wird in Abschnitt 7.3 beschrieben.

Es fehlt nun noch die Beschreibung des gewünschten bzw. erwarteten Verhaltens nach Ausführung des aufgerufenen Protokolls. Hierbei muss man zunächst unterscheiden, ob das aufgerufene Sequenzdiagramm ohne Fehler durchlaufen oder ob eine Exception geworfen wurde. Ein Fehler tritt zum Beispiel dann auf, wenn während der Ausführung ein UML-`FlowFinalNode`

erreicht wird (modelliert in dem zugehörigen Aktivitätsdiagramm). Auch einige in MEL vordefinierte Operationen werfen Exceptions, zum Beispiel beim Entschlüsseln eines Datums mit einem falschen Schlüssel. Tritt während des Ausführens eines Protokolls eine Exception auf, wird der Protokollschritt, in dem der Fehler auftritt, abgebrochen. Als Folge davon wird in diesem Protokollschritt keine Nachricht versendet und somit finden die nachfolgenden Protokollschritte des Protokolls ebenfalls nicht statt. Eine UML-`Action`, die einen Sequenzdiagrammaufruf enthält, hat in der Regel zwei ausgehende Kanten, d.h. der Kontrollfluss verzweigt an dieser Stelle. Der eine Zweig gibt an, wie sich die Anwendung bei einem erfolgreichen, d.h. fehlerfreien, Durchlauf des Protokolls verhalten sollte und der andere Zweig beschreibt die durchzuführenden Tests, d.h. das erwartete Verhalten, im Fehlerfall.

Es muss nun noch erläutert werden, wie der erwartete Zustand der Komponenten nach Auftreten eines Fehlers sowie im Falle eines fehlerfreien Durchlaufs eines Protokolls modelliert wird.

4. Die UML-`Action`, die einen Sequenzdiagrammaufruf enthält, hat mindestens einen und maximal zwei ausgehende Kanten. Ein ausgehender UML-`ControlFlow` modelliert den erwarteten Zustand des Systems, wenn das zuvor aufgerufene Sequenzdiagramm erfolgreich durchlaufen wurde (4a). Dies ist der Fall, wenn bei Ausführung des Protokolls oder Angriffsszenarios keine Exception geworfen wurde. Ein ausgehender UML-`ExceptionHandler` modelliert den Zustand bei Auftreten eines Fehlers während der Protokollausführung (4b). Hat eine UML-`Action` zwei ausgehende Kanten, ist eine davon ein UML-`ControlFlow`, der andere ein UML-`ExceptionHandler`. Das Ende eines UML-`ExceptionHandlers` zeigt auf ein UML-`InputPIN`, das sich an einer UML-`Action` befindet.
5. Um den Zustand der modellierten Anwendung nach Ausführung eines Protokolls zu beschreiben, können UML-`DecisionNodes` für Verzweigungen (bei denen die Verzweigungsbedingungen durch einen booleschen MEL-Ausdruck modelliert sind) und UML-Actions verwendet werden. Dies ist sowohl für die Beschreibung des Zustands im Fehler- als auch im Nichtfehlerfall möglich. Die Verwendung von Verzweigungen und Actions nach dem Protokollaufruf ist optional.
6. Beim Erstellen des Testfalldiagramms erwartet der Modellierer entweder, dass ein Protokoll ohne Fehler durchläuft (in der Regel ist dies bei funktionalen Tests der Fall) oder dass beim Durchlaufen des Protokolls ein Fehler auftritt (z.B. bei Modellierung eines Angriffs, der natürlich fehlschlagen sollte). Ein UML-`ActivityFinalNode` markiert das erwartete Ende eines Testfalls, d.h. der Modellierer erwartet, dass dieser Knoten bei Ausführung des Testfalls erreicht wird (6a). Ein UML-`FlowFinalNode` markiert das Ende eines Testfalls, das von dem Modellierer nicht erwartet wird (6b). Beispielsweise, wenn unerwarteterweise eine Exception bei Ausführung des Protokolls auftritt. Wird bei Ausführung des Testfalls ein UML-`FlowFinalNode` erreicht, schlägt der zugehörige Test fehl.

Zusätzlich soll ein Testfall weitere Bedingungen überprüfen und melden, wenn diese nicht eingehalten werden. Stimmt eine Bedingung während oder nach einem Protokolllauf nicht, schlägt der Test fehl. Um die von der Anwendung einzuhaltenden Bedingungen zu beschreiben, werden UML-Constraints verwendet. Ein Constraint kann einen beliebigen Text enthalten. Oft werden Constraints in UML mit der Object Constraint Language (OCL, [74]) angegeben.

Für die Angabe der Constraints in den Testfällen wird die Sprache MEL verwendet. Da sich die Constraints immer auf einzelne Komponentenklasseninstanzen beziehen und MEL durch die plattformunabhängige Modellierung den Entwicklern bereits bekannt ist, ist die Sprache hierfür gut geeignet. Ein Constraint kann für eine UML-`Action`, die einen Protokollaufruf enthält, sowie alle folgenden UML-Elemente angegeben werden.

7.2.1.1. Initialisierung von Komponenten und Festlegung konkreter Eingabewerte

In dem Testfall für das Abfragen des Kontostands wurden für das Erzeugen der entsprechenden Instanzen der Komponentenklassen parameterlose Konstruktoren verwendet. Bei diesen werden alle Attribute und Assoziationen der Instanzen mit Defaultwerten initialisiert, z.B. erhalten Schlüsselpaare feste, zueinander passende Schlüssel. Weiterhin ist es möglich, die Konstruktoren aller Klassen mit Argumenten aufzurufen. Die Argumente ergeben sich dabei aus den Attributen und Assoziationsenden, die im plattformunabhängigen Klassendiagramm mit ≪Initialize≫ annotiert sind. Für das Erzeugen von Objekten und die Konstruktoraufrufe gelten dieselben Regeln wie bei der plattformunabhängigen Modellierung (siehe Abschnitt 4.2.1.7).

Ein Problem ist das mögliche Auftreten von Namenskonflikten. Diese kommen zustande, weil bei der Generierung des Quellcodes für jede Komponentenklasse ein eigenes Java-Package erzeugt wird, das den Quellcode für diese Klasse enthält. Viele Klassen, beispielswiese die vordefinierten Sicherheitsdatentypen, sind in jedem der Java-Packages vorhanden. Der Grund ist, dass diese Klassen von allen Komponenten verwendet werden. Zum Beispiel ist die Klasse `Secret` sowohl im Package `DepositMachine` als auch im Package `CopyingMachine` vorhanden, weil beide Komponenten ein Attribut vom Typ `Secret` besitzen. Deshalb muss bei der Deklaration von lokalen Variablen zusätzlich zur Typangabe angegeben werden, für welches Package die lokale Variable erzeugt wird. Für die Komponentenklassen ist dies nicht notwendig, da diese nur in einem Java-Package definiert sind.

Abbildung 7.3 zeigt die Initialisierung der Komponentenklassen der Kopierkartenanwendung und erläutert gleichzeitig die Auflösung der Namenskonflikte.

```
secretT : DepositMachine.Secret := create  DepositMachine.Secret("12345678");
secretCM : CopyingMachine.Secret := create  CopyingMachine.Secret("12345678");
secretC : Copycard.Secret := create Copycard.Secret("12345678");
user : CardOwner := create CardOwner();
card : Copycard := create Copycard(0, secretC, Copycard.IDLE_CARD);
terminalDM : DepositMachine := create DepositMachine(secretT, DepositMachine.IDLE_TERMINAL);
terminalCM : CopyingMachine := create CopyingMachine(secretCM, CopyingMachine.IDLE_TERMINAL);
```

Abbildung 7.3.: Initialisierung für die Tests der Kopierkartenanwendung

Die Ladeterminals, die Kopiergeräte sowie die Kopierkarten benötigen alle ein gemeinsames Geheimnis vom Typ `Secret`. Aus diesem Grund sollen alle drei Komponenten mit dem Geheimnis „12345678“ initialisiert werden. Die Klasse `Secret` ist im generierten Code in den drei Java-Packages `DepositMachine`, `CopyingMachine` und `Copycard` vorhanden.

Es müssen deshalb drei lokale Variablen deklariert werden, für jedes Java-Package eine. Der MEL-Ausdruck `secretT : DepositMachine.Secret` erzeugt eine lokale Variable mit Namen `secretT` vom Typ `DepositMachine.Secret`. `DepositMachine` ist der Name des Java-Packages. Anschließend werden die vier Instanzen `user`, `card`, `terminalDM` und `terminalCM` erzeugt.

Die Initialisierung der Komponenten kann in ein separates Aktivitätsdiagramm ausgelagert werden, das nur eine UML-`Action` mit entsprechenden MEL-Ausdrücken enthält. Dieses Aktivitätsdiagramm kann dann von verschiedenen Testfalldiagrammen aus referenziert werden (siehe Abbildung 7.4). Auf diese Weise muss die Initialisierung nur einmal modelliert werden.

Erwartet das zu testende Protokoll Eingabewerte von der aufrufenden User-Instanz, müssen diese ebenfalls definiert werden. Zum Beispiel muss beim Aufladen der Kopierkarte angegeben werden, welcher Betrag auf die Karte geladen werden soll. Die im plattformunabhängigen Modell definierte Klasse `CardOwner`, die den Kartenbesitzer modelliert, besitzt entsprechende Attribute für die zu tätigenden Eingaben. Im Beispiel der Kopierkartenanwendung ist dies das Attribut `value`, das im Aktivitätsdiagramm des Bezahlprotokolls verwendet wird. Um festzulegen, welcher Wert in einem konkreten Testfall auf die Karte geladen werden soll, muss das Attribut `user.value` entsprechend gesetzt werden. Dies muss jedoch im Testfalldiagramm (in einer UML-`Action` nach Durchführung der Initialisierung) und nicht im ausgelagerten Protokoll für die Initialisierung der Komponenten geschehen, da die konkreten Eingabewerte von Testfall zu Testfall unterschiedlich sind.

7.2.1.2. Protokollaufrufe

Die im Anwendungsmodell definierten Protokolle können im Testfalldiagramm aufgerufen werden. Diese Aufrufe beziehen sich immer auf die Sequenzdiagramme der Protokolle. Der Einheitlichkeit wegen gilt dies sowohl für den Aufruf bestehender Protokolle als auch den Aufruf von Angriffsszenarien. Die Sequenzdiagramme sind in der Regel kürzer und übersichtlicher, die Aktivitätsdiagramme hingegen beschreiben die Protokolle vollständig. Ein Angreifer hat im SecureMDD-Ansatz eine Black Box-Sicht auf die Anwendung, d.h. er kann die Kommunikation zwischen den Komponenten beeinflussen. Diese ist sowohl im Sequenz- als auch im Aktivitätsdiagramm modelliert. Die Berechnungen der einzelnen Komponenten kann der Angreifer hingegen nicht manipulieren, da er keinen Zugriff auf die Komponenten selber bzw. ihren Speicher hat. Sequenzdiagramme reichen somit aus, um mögliche Angriffe zu formulieren. Sie sind zusätzlich übersichtlicher und schneller zu erstellen als Aktivitätsdiagramme.

Der Aufruf eines Sequenzdiagramms geschieht durch Angabe des entsprechenden Diagramms im Behavior-Feld der UML-`Action`. Es ist auch möglich, mehrere Protokolle hintereinander aufzurufen. Hierfür wird für jeden Aufruf eine eigene UML-`Action` verwendet.

7.2.1.3. Angabe von Invarianten sowie von (Nach-)bedingungen

Eine Invariante wird durch einen booleschen MEL-Ausdruck angegeben und ist in einem UML-Constraint definiert. Dieser Constraint muss zu einer UML-`Action`, die ein Protokoll aufruft, gehören. Die Invariante muss zu jedem Zeitpunkt, d.h. vor dem Protokolllauf sowie nach jedem Protokollschritt, erfüllt sein, ansonsten schlägt der Test fehl. Sind mehrere Invarianten angegeben, müssen alle erfüllt sein.

Für einige Testfälle möchte man Bedingungen angeben, die nach Ausführung des Protokolls gelten müssen. Diese werden, ebenfalls in Form von booleschen MEL-Ausdrücken, als Constraint formuliert und können an den UML-`ControlFlows` und UML-`Actions`, die sich hinter dem Protokollaufruf befinden, modelliert werden. Hinter bedeutet in diesem Fall, dass die UML-Elemente nach Ausführung des Protokolls „durchlaufen“ werden. Die so formulierten Bedingungen werden nur einmal, und zwar nach Ausführung des Protokolls, ausgewertet. Durch die Verwendung von UML-`DecisionNodes` sowie `Guards` kann noch weiter eingeschränkt werden, wann (d.h. in welchem Zustand der Komponenten) die Bedingungen überprüft werden sollen.

Abbildung 7.4 zeigt als Beispiel ein Testfalldiagramm für das Bezahlen mit der Kopierkarte. Der Test soll sicherstellen, dass nur dann Geld von der Karte abgebucht wird, wenn noch genügend Guthaben vorhanden ist. Außerdem wird getestet, dass der korrekte Betrag abgebucht wird.

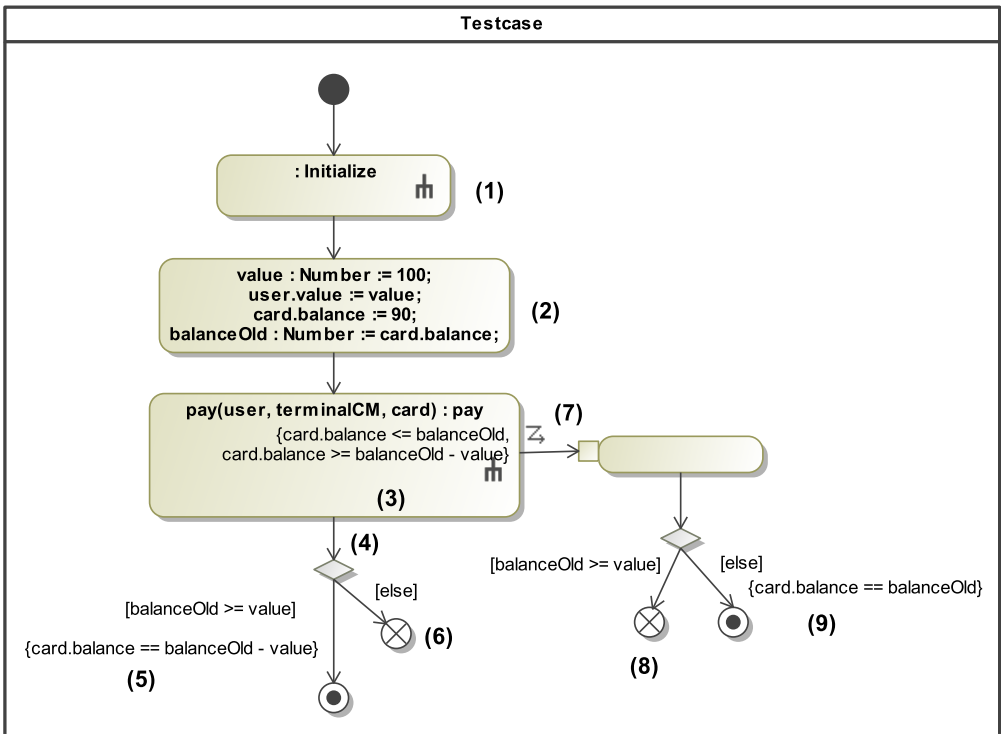

Abbildung 7.4.: Testfalldiagramm für das Bezahlen mit der Kopierkarte

Für die Initialisierung der Komponenten wird das in Abbildung 7.3 dargestellte Aktivitätsdiagramm mit dem Namen `Initialize` aufgerufen (1). Im Anschluss daran werden noch weitere Initialisierungen vorgenommen (2). Die lokale Variable `value` wird deklariert und mit dem Wert 100 (Cent) initialisiert. Anschließend wird der Wert dieser Variablen dem Attribut `value` des `CardOwners` `user` zugewiesen. Dieses Attribut wird für die Eingabe des Protokolls zum Anfertigen von Kopien verwendet. Weiterhin wird der Kontostand der Instanz der Klasse `Copycard` auf 90 (Cent) gesetzt, d.h. auf der für den Test verwendeten Kopierkarte befinden sich nun 90 (Cent). Abschließend wird dieser Kontostand in der lokalen Variablen `balanceOld` gespeichert. Das Protokoll wird nun mit den Instan-

zen `user`, `terminalCM` und `card` aufgerufen (3). Konkret bedeutet dies, dass versucht wird, von einer Karte, auf der 90 (Cent) gespeichert sind, 100 (Cent) abzubuchen. Für den Protokollaufruf sind zwei Constraints angegeben (`card.balance <= balanceOld`, `card.balance >=balanceOld-value`). In diesem Beispiel sollte der Kontostand entweder gleich `balanceOld` (bevor der Wert `value` abgebucht wurde) oder gleich `balanceOld - value` (nach Abbuchung von `value`) sein. Dies ist durch das angegebene Intervall beschrieben.

Der Kontrollfluss nach unten im Diagramm beschreibt den Fall, dass das Protokoll ohne Fehler durchlaufen wird (4). Für den Fall, das `balanceOld` größer oder gleich `value` ist (d.h. die Karte hat ausreichend viel Guthaben für die Abbuchung, was bei der angegebenen Initialisierung nicht erfüllt ist), sollte der Betrag `value` von der Karte abgebucht werden. Dies ist durch einen entsprechenden Constraint formuliert (5). Ist `balanceOld` kleiner als `value` (d.h. der Kontostand ist zu niedrig für die aktuelle Abbuchung), soll eigentlich während des Protokolllaufs eine Exception geworfen werden. Wird der `else`-Fall dennoch erreicht, deutet dies auf einen Fehler hin und der Test schlägt fehl. Dies ist durch einen UML-`FlowFinalNode` modelliert (6).

Der Kontrollfluss, der von dem Protokollaufruf ausgehend, nach rechts weggeht, beschreibt den erwarteten Zustand der Komponenten nach Auftreten eines Fehlers während des Protokolllaufs (7). Dies ist durch einen `ExceptionHandler` modelliert. Ist `balanceOld` in diesem Fall größer oder gleich `value`, wurde die Exception während des Protokolllaufs nicht aufgrund des nicht gedeckten Kontostandes, sondern aus einem anderen Grund geworfen. Dies könnte ein Hinweis auf einen (logischen) Protokollfehler sein und der Test soll fehlschlagen. Dies ist durch ein UML-`FlowFinalNode` modelliert (8). Ist `balanceOld` kleiner als `value`, soll der Kontostand nicht verändert werden. Dies ist in dem Constraint `card.balance == balanceOld` formuliert (9). In unserem konkreten Beispiel wird dieser Zweig ausgeführt (d.h. während des Protokolllaufs wird eine Exception geworfen und die Bedingung `balanceOld < value` gilt). Der Constraint ist nach Ausführung des Protokolls erfüllt und der Test somit erfolgreich.

Es fehlt nun noch die Modellierung von Angriffen. Diese wird im folgenden Abschnitt erläutert.

7.3. Modellierungsrichtlinien für die Definition eines Angriffsszenarios

Beim Entwurf der Modellierungsrichtlinien für die Definition eines Angriffsszenarios stand die intuitive Modellierung im Vordergrund. Das Ziel war es, dass die Angriffsszenarien leicht zu erstellen, übersichtlich sowie für eine außenstehende Person intuitiv verständlich sind. Bei den mit SecureMDD entwickelten Anwendungen wird von einem Angreifer ausgegangen, der lesend und schreibend Zugriff auf die Kommunikationskanäle hat sowie gesendete Nachrichten unterdrücken kann. Das Ziel war es deshalb, die Modellierung und Durchführung von Protokollangriffen, die von einem solchen Angreifer durchgeführt werden können, zu ermöglichen. Die Modellierungsrichtlinien unterstützen deshalb die Modifikation, das Umleiten, das Unterdrücken sowie das Wiedereinspielen (Replay) von Nachrichten, die zwischen zwei Komponenten ausgetauscht werden. Die verwendete Modellierung verzichtet dabei auf die explizite Modellierung eines Angreifers sowie seines Wissens. Dies ist konform zu der Modellierung der

Anwendung, bei der der Angreifer ebenfalls nur implizit, über die Annotationen im Deploymentdiagramm, vorhanden ist. Ein Angriffsszenario wird durch ein Sequenzdiagramm modelliert. Dieses muss sich auf ein im Anwendungsmodell definiertes Protokoll beziehen. In der Regel bietet es sich an, hierfür die bestehenden Sequenzdiagramme der Protokolle zu kopieren und abzuändern.

Im Folgenden werden die Richtlinien zur Modellierung von Angriffen erläutert.

7.3.1. Manipulation und Wiedereinspielen von Nachrichten

Um anzugeben, dass eine Nachricht vom Angreifer manipuliert wird, wird diese Nachricht im Sequenzdiagramm abgeändert. Dies ist möglich, indem die Argumente der Nachricht verändert werden. Sollen Argumente gleich bleiben, wird dies durch einen Unterstrich „_“ angegeben. Abbildung 7.5 (links) zeigt die Manipulation der `Pay`-Nachricht. Dort wird das erste Argument (der von der Karte abzubuchende Betrag) auf 0 gesetzt, während das zweite Argument (der Hashwert) gleich bleibt. Die Reihenfolge der Argumente ergibt sich, wie bei der plattformunabhängigen Modellierung, aus dem Klassendiagramm der Anwendung.

Natürlich können nicht nur einzelne Teile, sondern auch eine komplette Nachricht ausgetauscht werden. In diesem Fall wird der Name der Nachricht durch den Namen einer anderen Nachricht ersetzt. In Abbildung 7.5 (links) würde dies bedeuten, dass die Nachricht `Pay` durch eine andere Nachricht, zum Beispiel `RequestBalance`, ersetzt wird.

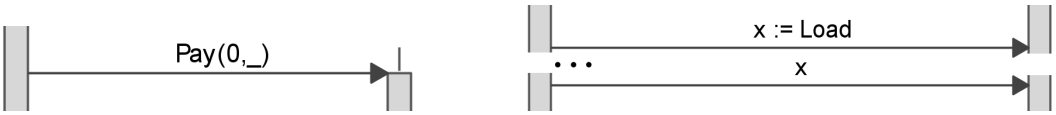

Abbildung 7.5.: Modifizieren (links) sowie Wiedereinspielen (rechts) von Nachrichten

Das Wiedereinspielen von abgehörten Nachrichten zu einem späteren Zeitpunkt ist ebenfalls möglich. Hierfür ist es zunächst notwendig, die Nachricht, die später erneut gesendet werden soll, zwischenzuspeichern. Dies ist möglich, indem die Nachricht an eine lokale Variable zugewiesen wird. Diese Variable muss nicht explizit deklariert werden, ihr Typ ergibt sich automatisch aus der ihr zugewiesenen Nachricht. Diese Variable kann dann in den folgenden Schritten verwendet werden, um die Nachricht erneut zu senden und auf diese Weise Replay-Angriffe zu realisieren. In Abbildung 7.5 (rechts) wird die Nachricht `Load` in der Variablen `x` zwischengespeichert und später erneut gesendet.

7.3.2. Umleiten und Unterdrücken von Nachrichten und die Verwendung komplexer Ausdrücke

Jede Nachricht kann an eine andere Lebenslinie im Sequenzdiagramm umgeleitet werden. Dies wird erreicht, indem die umzuleitende Nachricht auf eine andere Lebenslinie zeigt als im Originalprotokoll. Ein Beispiel hierfür zeigt Abbildung 7.6 (links). Außerdem kann eine Nachricht unterdrückt werden, indem sie aus dem Sequenzdiagramm gelöscht wird. In diesem Fall muss allerdings auch die zugehörige Antwortnachricht aus dem Diagramm entfernt werden.

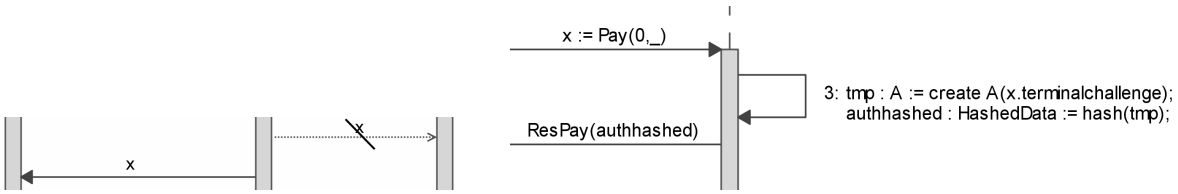

Abbildung 7.6.: Umleiten von Nachrichten (links) sowie die Verwendung komplexer Ausdrücke (rechts)

Um auch komplexere Manipulationen von Nachrichten oder das Erstellen neuer Nachrichten modellieren zu können, ist die Verwendung von MEL-Ausdrücken in den Sequenzdiagrammen möglich. Diese werden durch die Verwendung von self-Nachrichten, d.h. dem Versenden einer Nachricht an die eigene Lebenslinie, angegeben. So ist es dem Modellierer möglich, neue Objekte zu erzeugen oder in MEL definierte kryptographische Operationen, z.B. für die Verschlüsselung oder das Bilden eines Hashwertes, aufzurufen. Abbildung 7.6 (rechts) zeigt ein Beispiel hierfür. Dort wird ein Objekt der Klasse `A` unter Verwendung der zuvor gespeicherten Nachricht `Pay` erstellt und anschließend der Hashwert gebildet. Das Ergebnis wird als Argument der `ResPay`-Nachricht verschickt.

Beispiel für einen modellierten Angriff

Ein Angriff auf das Protokoll zum Aufladen der Kopierkarte könnte das wiederholte Senden der `Load`-Nachricht an die Kopierkarte sein. Ein fehlerhaftes Protokoll könnte diese Nachricht erneut akzeptieren und den Kontostand der Karte wiederholt erhöhen, obwohl der Betrag nur einmal in das Ladeterminal eingeworfen wurde. Dieser Angriff sowie das zugehörige Testfalldiagramm sind in Abbildung 7.7 dargestellt.

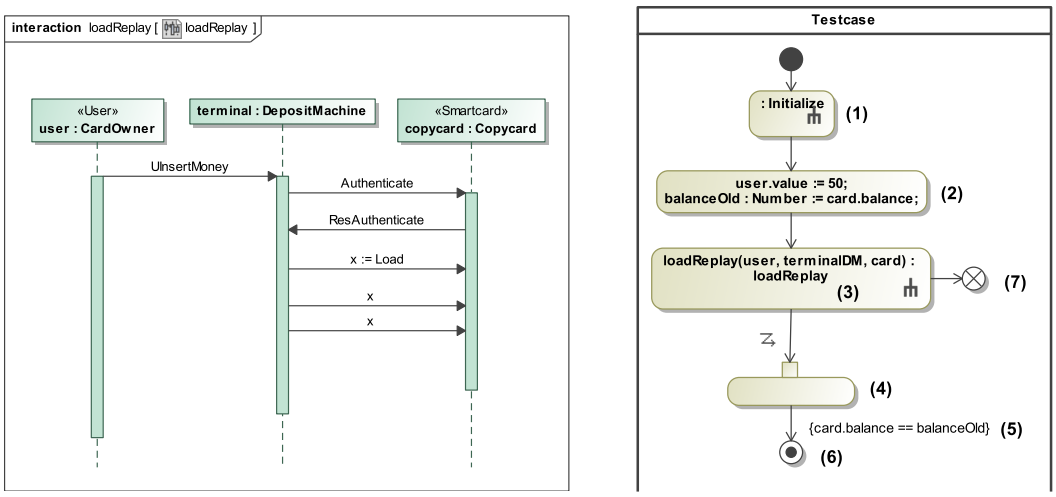

Abbildung 7.7.: Replay-Angriff (links) sowie zugehöriges Testfalldiagramm (rechts)

Der Angriff bezieht sich auf das Protokoll zum Aufladen der Kopierkarte. Diese Zuordnung geschieht durch den Namen der Benutzernachricht, die den Protokolllauf anstößt, in diesem Beispiel die Nachricht `UInsertMoney`. Die ersten drei Nachrichten stimmen mit denen des

Originalprotokolls überein. Im vierten Schritt wird die Nachricht `Load` an die Kopierkarte gesendet und gleichzeitig in der lokalen Variablen `x` zwischengespeichert und nachfolgend noch zweimal an die Karte gesendet.

Das Testfalldiagramm (in Abbildung 7.7 rechts zu sehen) ruft dieses Angriffsdiagramm auf. Zunächst werden die Komponentenklassen instanziiert. Hierfür wird das Aktivitätsdiagramm `Initialize` aufgerufen (1), das in Abbildung 7.3 vorgestellt wurde. Im Anschluss wird das Attribut `value`, das den auf die Karte zu ladenden Betrag speichert, auf 50 gesetzt und der aktuelle Kontostand der Kopierkarte in der Variablen `balanceOld` zwischengespeichert (2). Dann wird das links im Bild dargestellte Angriffsszenario aufgerufen (3). Dieses hat den Namen `loadReplay`. Der Angriff soll natürlich nicht erfolgreich sein und somit während des Protokolllaufs ein Fehler auftreten, d.h. eine Exception geworfen werden. Aus diesem Grund folgt dem modellierten `ExceptionHandler` ein `ActivityFinalNode`. Wird dieser erreicht (4) und ist gleichzeitig die Bedingung `card.balance == balanceOld + value` (5) erfüllt (d.h. die Karte wurde nur einmal aufgeladen), ist der Test erfolgreich (6). Ist die Bedingung nicht erfüllt oder läuft das Protokoll ohne Auftreten einer Exception durch, schlägt der Test fehl (7).

Nachdem die Modellierung von funktionalen Tests und Angriffsszenarien erläutert wurde, mag die Frage aufkommen, weshalb die Testfälle nicht automatisch anhand der modellierten Anwendungsprotokolle erstellt werden. Es ist, auch wenn dies nicht ganz einfach zu realisieren ist, möglich, funktionale Testfälle aus den Diagrammen abzuleiten. Die benötigten Initialisierungen und Eingabewerte könnten entweder im Modell angegeben oder aus einer Datenbank ausgelesen werden. Es wurde trotzdem ein Weg gewählt, bei dem die Testfälle explizit modelliert werden müssen. Dies hat den Vorteil, dass z.B. ganz konkret verschiedene Invarianten und Nachbedingungen für vorgegebene Eingaben getestet und „durchprobiert“ werden können. Außerdem ist es auf diese Weise möglich, die Testfälle sowie die benötigten Initialisierungen flexibel anzugeben und dabei auch komplexere Zusammenhänge mit MEL zu formulieren. Für das Testen einer Anwendung auf Sicherheit ist die automatische Generierung von Testfällen außerdem nicht mehr ganz so einfach. Insbesondere ist nicht klar, nach welchen Kriterien Testfälle generiert werden können. Es ist wohl realisierbar, einige generische, vordefinierte Angriffe zu konstruieren und zu testen. Der Test einer Anwendung auf anwendungsspezifische Eigenschaften ist auf diese Weise jedoch nicht möglich.

Die Unterstützung von automatisch generierten, funktionalen Testfällen bzw. entsprechenden Eingabedaten (Fuzz-Testing) sowie das Einbinden einer Bibliothek mit Standard-Angriffsszenarien ist jedoch eine sinnvolle Ergänzung des bestehenden Testframeworks. Weiterhin ist das Hinzufügen von statischen Checks, die einzelne Codefragmente testen, hilfreich. Auf diese Weise werden auch Fehler gefunden, die unabhängig von den modellierten Protokollen und der konkreten Anwendung sind (z.B. die Implementierung der Listenoperationen).

7.4. Codegenerierung für Testfälle

In diesem Abschnitt wird auf die bei der Implementierung der Modelltransformationen getroffenen Designentscheidungen eingegangen. Detailliert ist die Generierung in [105] beschrieben.

Bei der Generierung des Testfallcodes standen zwei Designprinzipien im Vordergrund:

- Der Anwendungscode und der Testfallcode sollen möglichst unabhängig voneinander bleiben. Diese Trennung hat den Vorteil, dass der getestete Anwendungscode auch für den produktiven Einsatz verwendet werden kann, ohne dass er hierfür angepasst, d.h. Code, der nur für die Testfälle benötigt wird, entfernt werden muss.
- Der Testfallcode soll den Code testen, der später auch tatsächlich auf den Smart Cards und Terminals läuft.

Der erste Punkt verlangt, dass der produktiv eingesetzte Code frei von Testartefakten ist. Der zweite Punkt hingegen setzt voraus, dass der Testcode Zugriff auf die internen Zustände der Komponenten hat und den eigentlich vorgesehenen Protokollablauf entsprechend der Angriffsszenarien beliebig ändern kann. Diese Anforderungen sind widersprüchlich.

Dieser Widerspruch wird durch eine Instanz des *Strategie-Patterns* [59] aufgelöst. Ein Interface, genannt Gate, definiert Methoden, über die Constraint-Prüfungen und Manipulationen der Protokollabläufe durchgeführt werden können. Der generierte Anwendungscode ruft Methoden dieses Interfaces an genau definierten Stellen auf. Dabei wird bei produktiv laufendem Code eine Defaultimplementierung des Gates verwendet, die die im Anwendungsmodell definierten Protokollabläufe nicht beeinflusst. Während des Testens der Anwendung wird dagegen eine für jeden Testfall spezifische Implementierung des Gates verwendet. Die Stellen, an denen die Methoden des Gate-Interfaces aufgerufen werden, sind so gewählt, dass alle modellierbaren Testfälle und Angriffsszenarien abgebildet werden können.

7.5. Verwandte Arbeiten

Modellbasiertes Testen ist ein Forschungsbereich, der in den letzten Jahren viel Beachtung fand. Dementsprechend vielseitig sind die erreichten Ergebnisse und Ansätze zu dem Thema. Keine der bestehenden Arbeiten unterstützt jedoch das Modellieren von funktionalen Tests und möglichen Angriffszenarien mit UML bei gleichzeitiger Generierung von lauffähigem Testcode, um den (ebenfalls generierten) Anwendungscode zu testen.

Bouquet et al. stellen in [27] die Modellierung einer Anwendung mit UML und OCL vor. Aus diesem Anwendungsmodell können automatisch Testfälle abgeleitet werden. Das Testen von Sicherheitsprotokollen ist jedoch nicht Bestandteil des Ansatzes.

Sensler et al. [174] haben einen Ansatz entwickelt, der den Entwurf und Generierung von funktionalen Testfällen für Webanwendungen unterstützt. Der Ansatz erlaubt jedoch nur die Modellierung von funktionalen Tests, das Testen einer Anwendung auf Sicherheit ist nicht möglich. Dennoch sind die SecureMDD-Modellierungsrichtlinien für Testfälle inspiriert von diesem Ansatz. Insbesondere die Definition wiederverwendbarer Unterprozeduren und die Idee des modularen Aufbaus der Testfälle wurden aus [174] übernommen.

Fourneret et al. [56] modellieren eine Smart Card-Anwendung mit UMLSec und definieren Sicherheitseigenschaften auf diesem Modell. Die von automatischen Theorembeweisern auf einem abstrakten Modell verifizierten Eigenschaften werden verwendet, um Testfälle für den Anwendungscode zu generieren. Der Ansatz unterstützt jedoch nicht die Modellierung von möglichen Angriffen auf die entworfenen kryptographischen Protokolle.

Jürjens et al. stellen in [97, 101] ihren werkzeugunterstützten Ansatz für die systematische Generierung von Testfällen für die Implementierung einer formal spezifizierten Anwendung

vor. Der Entwurf eines eigenen Modellierungsansatzes für Testfälle, wie er in diesem Kapitel vorgestellt wurde, existiert jedoch nicht. Dies wird als geplante Erweiterung genannt.

In [6] diskutieren Apel et al. ein modellbasiertes Verfahren zur Testfallgenerierung. Dies wird am Beispiel der Elektronischen Gesundheitskarte evaluiert. Die zu testende Chipkartenanwendung wird mittels asynchroner Produktautomaten modelliert. Aus diesen lassen sich abstrakte Kommandofolgen berechnen, die in konkrete ausführbare Kommandos (APDUs) übersetzt werden. Ziel ist es, Implementierungsfehler zu finden und gleichzeitig eine hohe Testabdeckung zu erreichen. Das Testen auf Sicherheitslücken ist mit dem Ansatz jedoch nicht möglich. Stattdessen wird die Korrektheit der einzelnen Kommandos betrachtet.

Morais et al. stellen in [138] vor, wie man aus Angriffsbäumen (Attack Trees, [172]) Testszenarien ableiten kann, die die Implementierung eines Sicherheitsprotokolls testen. Als Angreifermodell wird ein Dolev Yao-Angreifer [51] vorausgesetzt, der vollen Zugriff (Lesen, Schreiben, Unterdrücken von Nachrichten) auf alle Kommunikationskanäle hat. Um die in natürlicher Sprache erstellten Angriffsbäume transformieren zu können, sind noch manuelle Verfeinerungsschritte nötig. In diesen wird jedes Angriffsziel durch ein Angriffsmuster (Angriffsziel, Vorbedingungen, Schritte des Angreifers) beschrieben. Als Fallstudie wird ein auf TLS aufbauendes Kommunikationsprotokoll verwendet. Das Durchführen von funktionalen Tests sowie die Modellierung eines konkreten Angriffsszenarios für kryptographische Protokolle (wie es in dieser Arbeit beschrieben wird), ist mit dem in [138] vorgestellten Ansatz nicht möglich. Dennoch ist die Idee, Testfälle anhand von Angriffsbäumen zu generieren, auch für den SecureMDD-Ansatz interessant.

Da das Testen von kryptographischen Protokollen spezielle Herausforderungen birgt, sind Ansätze zum Testen anderer Arten von sicherheitskritischen Anwendungen (z.B. für Access Control Policies [157]) nicht auf diesen Problembereich übertragbar und werden deshalb hier nicht vorgestellt.

Das Ziel der in diesem Kapitel vorgestellten Arbeiten ist das Testen auf Sicherheitslücken, die durch (logische) Fehler in den zugrundeliegenden Sicherheitsprotokollen entstehen. Das Finden von z.B. Buffer Overflows oder SQL-Injection-Angriffen steht nicht im Fokus dieser Arbeit. Ansätze wie z.B. [154] sind deshalb nicht vergleichbar mit dem modellbasierten Testen in SecureMDD.

Der in [184] von Tsankov et al. vorgestellte Ansatz verwendet die Technik des Fuzz-Testings, bei dem zufällige Eingabedaten zum Testen verwendet werden. Wie schon erwähnt ist das Fuzz-Testing auch für den SecureMDD-Ansatz interessant und stellt eine mögliche Erweiterung des Ansatzes dar.

Weitere Ansätze zum modellbasierten Testen (wie z.B. [7, 31]) basieren auf Model-Checking. Die Verwendung von Model-Checkern zur Validierung und Verifikation wird in Abschnitt 8.5 diskutiert.

Teil IV.

Verifikation von Sicherheitsaussagen sowie Korrektheit des Codes

8 Generierung der formalen Spezifikation

Zusammenfassung: Aus dem plattformunabhängigen UML-Modell einer Anwendung lässt sich automatisch, durch Ausführung von Modell-zu-Text-Transformationen, ein formales Modell generieren. Dieses basiert auf algebraischen Spezifikationen und Abstract State Machines (ASM) [25, 77]. Die generierte formale Spezifikation ist auf die Entwicklung von Anwendungen, die auf kryptographischen Protokollen basieren, zugeschnitten und wird verwendet, um Sicherheitseigenschaften für die modellierte Anwendung zu verifizieren. Gleichzeitig beschreibt die formale Spezifikation die Semantik der UML-Modelle. Das heißt, die Übersetzungsregeln, die aus einem plattformunabhängigen UML-Modell das formale Modell erzeugen, definieren die Semantik des plattformunabhängigen UML-Anwendungsmodells. In diesem Kapitel werden das formale Modell sowie die Transformation des UML-Modells in die formale Spezifikation erläutert. Dabei werden die Transformationen informell sowie anhand von Beispielen beschrieben. Die Implementierung der Transformationen ist in [128] veröffentlicht. Die Generierung der formalen Spezifikation ist außerdem in [135, 136] publiziert.

In Abschnitt 8.1 werden die in diesem Kapitel benötigten Grundlagen erläutert. Abschnitt 8.2 skizziert den Aufbau und die Funktionsweise des formalen Modells. Abschnitt 8.3 gibt einen Überblick über den Prosecco-Ansatz [78], auf dem der formale Teil dieser Arbeit aufbaut bzw. der in dieser Dissertation signifikant verbessert wurde, und erläutert die Unterschiede. Abschnitt 8.4 beschreibt die Transformationsregeln, mit denen aus einem UML-Modell die formale Spezifikation erzeugt wird und Abschnitt 8.5 erläutert verwandte Arbeiten. Die formale Spezifikation der Kopierkartenanwendung ist vollständig im Anhang D angegeben.

8.1. Grundlagen

8.1.1. Gentzen Beweiskalkül

Grundlage für die in dieser Arbeit geführten Beweise ist ein auf Gentzen zurückgehender Sequenzenkalkül [53]. Dieser basiert auf Sequenzen der Form $\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$, die eine vereinfachende Schreibweise für $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \psi_1 \vee \ldots \vee \psi_m$ sind. Im Folgenden wird anstatt $\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$ auch $\Gamma \vdash \Delta$ als abkürzende Schreibweise verwendet.

Beweise im Sequenzenkalkül werden durch Anwendung von Beweisregeln der Form

$$\frac{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

geführt. $\Gamma \vdash \Delta$ wird Conclusio genannt und stellt das eigentliche Beweisziel dar. $\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n$ werden als Prämissen bezeichnet und stellen die resultierenden Beweisverpflichtung(en) dar. Aus den Prämissen folgt die Konklusion. Die Beweise werden aber üblicherweise rückwärts geführt.

Ein Beweis im Sequenzenkalkül wird durch wiederholte Anwendung von Beweisregeln geführt, bis keine offenen Prämissen mehr verbleiben. Dadurch entsteht ein Beweisbaum, bei dem die Wurzel die zu zeigende Aussage enthält und die Blätter jeweils durch Axiome gebildet werden.

Ein Beispiel für ein Axiom im Sequenzenkalkül ist, dass aus einer Vorbedingung eine identische Nachbedingung folgt:

$$\frac{}{\varphi \vdash \varphi}$$

Ein Beispiel für eine Regel mit mehreren Prämissen ist die für eine Fallunterscheidung der Disjunktion links:

$$\frac{\Gamma, \varphi \vdash \Delta, \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}$$

8.1.2. Algebraische Spezifikationen

Grundlage für die Spezifikation von Datentypen und Systemen in dieser Arbeit sind strukturierte algebraische Spezifikationen. Diese definieren Sorten sowie Konstanten, Variablen, Funktionen und Prädikate über diesen Sorten. Die Semantik der Konstanten, Funktionen und Prädikate wird mithilfe von Axiomen definiert. Lemmatas dienen dazu, abgeleitete Eigenschaften zu formulieren und zu beweisen.

Im KIV-System erlauben Basisspezifikationen die Definition beliebiger (nicht-frei erzeugter) Sorten mit zugehörigen Funktionen und Prädikaten. Neben diesen erlauben Datenspezifikation (`data specifications`) die Definition von frei erzeugten Datentypen. Nachfolgend sind zwei Beispiele für frei erzeugte Datentypen angegeben:

```
literature =
  mkbook (author : name; title : name )
| mkarticle (author : name; title : name; booktitle : name);

day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
```

Ein Element des Datentyps `literature` kann mithilfe der Konstruktoren `mkbook` und `mkarticle` erzeugt werden. Der Aufzählungstyp `day` besteht aus sieben Elementen, eines für jeden Wochentag.

Generische Spezifikationen haben einen Parameter-Datentyp, welcher mittels einer Aktualisierung durch einen konkreten Datentyp ersetzt werden kann. Damit ist es beispielsweise möglich, Datentypen wie Mengen einmalig zu definieren und dann konkrete Mengen von Zahlen oder Mengen von Buchstaben davon abzuleiten.

Anreicherungen sind Spezifikationen, die bereits bestehende Spezifikationen erweitern, beispielsweise indem Sortier-Funktionen auf Listen zugefügt werden. Vereinigungen erlauben es, die Sorten, Funktionen, Prädikate, Axiome und Lemmas mehrerer Spezifikationen zusammenzufügen.

8.1.3. Dynamische Higher Order Logik (DL)

Higher Order Logik [177] baut auf den algebraischen Datentypen auf und erweitert die Prädikatenlogik unter anderem um Variablen für Funktionstypen. Eine Funktionsvariable hat dabei die Typisierung $f : T_1 \times ... \times T_n \to T$. Ein Funktionsupdate hat die Form $f[x_1, ..., x_n, y]$ und es gilt $f[x_1, ..., x_n, y](x_1, ..., x_n) = y$. Statt der üblichen Form $f := f[x_1, ..., x_n, y]$ kann auch die vereinfachte Schreibweise $f(x_1, ..., x_n) := y$ verwendet werden.

Im Falle dynamischer Higher Order Logik (DL) [81,82] wird die Higher Order Logik noch um die Modaloperatoren [], $\langle\rangle$ und $\langle\!|\,|\!\rangle$ ergänzt, die imperative Programmkonstrukte enthalten. Die wichtigsten der in dieser Arbeit verwendeten Konstrukte sind im Folgenden beschrieben:

- Skip: `skip` stellt das leere Programm dar und terminiert ohne Zustandsänderung.
- Abort: `abort` ist das nicht-terminierende Programm.
- Zuweisung: `x := t` weist der Variablen `x` den Wert des Terms `t` zu.
- Parallele Zuweisung: $x_1 := t_1, ..., x_i := t_i$ führt alle Zuweisungen parallel aus. Aus Vereinfachungsgründen werden hier nur konfliktfreie parallele Zuweisungen betrachtet, bei denen die Variablen jeweils paarweise verschieden sind.
- Bedingte Verzweigung: `if` φ `then {`α`} else {`β`}` wertet zunächst die Bedingung φ aus. Gilt diese, so wird das Programm α ausgeführt, ansonsten das Programm β.
- Prozedurdeklaration: `P#(`$\underline{x}$`;` $\underline{y}$`) {`α`}` deklariert die Prozedur `P` mit den Eingabeparametern $\underline{x}$ und den Ein/Ausgabeparametern $\underline{y}$. Dabei können Eingabeparameter nur gelesen werden, Ein/Ausgabeparameter können sowohl gelesen als auch geschrieben werden.
- Prozeduraufruf: `P#(`$\underline{x}$`;` $\underline{y}$`)`
- Schleife: `while` φ `do {`α`}`
- Lokale Variablendeklaration: `var x=t in {`α`}`
- Sequentielle Hintereinanderausführung: α`;`β
- Indeterminismus: `choose x with` φ `in {`α`} ifnone {`β`}`: Eine lokale Variable `x` wird angelegt und mit einem zufälligen Wert belegt, für den jedoch die Bedingung φ zu wahr auswertet. Die angelegte Variable kann im Programm α verwendet werden.

Ein aus solchen Konstrukten bestehendes Programm kann dann mit den nachfolgenden Modaloperatoren in eine Sequenz eingefügt werden:

- $[\alpha]\varphi$ bedeutet, dass in jedem Fall, in dem das Programm α terminiert im Anschluss die Bedingung φ gilt.

- $\langle\alpha\rangle\varphi$ bedeutet, dass es mindestens einen terminierenden Ablauf gibt, nach dem die Bedingung φ gilt.
- $\langle\!|\alpha|\!\rangle\ \varphi$ bedeutet, dass alle Abläufe von α terminieren und im Endzustand jeweils φ gilt.

8.1.4. Abstract State Machines (ASM)

Abstrakte Zustandsmaschinen (Abstract State Machines, ASMs) [25, 77] sind ein Beschreibungsformalismus zu Spezifikation von Systemen. Die Beschreibung erfolgt anhand von Zustandsübergängen und setzt jeweils Mengen von Zuständen in Relation zueinander. Eine ASM wird definiert als ein Quadrupel `(S, Ini, Fin, R)`.

Dabei ist `S` eine Menge von Zuständen und `Ini` eine Teilmenge von `S`, die Anfangszustände. `Fin` beschreibt eine Teilmenge von `S`, die Endzustände und `R` eine Teilmenge von `S` × `S`, die Zustandsübergangsrelation. Die von der ASM erzeugten Abläufe sind genau die, welche mit einem Element aus `Ini` beginnen, durch fortgesetzte Anwendung von Übergängen aus `R` aufgebaut werden und deren letztes Element aus der Menge `Fin` ist. In einer ASM werden die Zustandsübergänge durch Regeln beschrieben. Ein Übergang entspricht der Ausführung einer ASM-Regel.

Die ASM-Regeln werden aus folgenden programmiersprachlichen Konstrukten gebildet:

- `f(p) := v`: Die dynamische Funktion `f` wird an der Stelle `p` aktualisiert, der neue Wert von `f` an dieser Stelle ist `v`.
- if φ `then` R_0 `else` R_1: Abhängig von der Auswertung der Formel φ wird entweder Regel R_0 oder Regel R_1 ausgeführt.
- `let x=t in R`: Eine lokale Variable `x` wird angelegt und dieser der Wert des Terms `t` zugewiesen. `x` bleibt nur im Kontext der Ausführung von `R` sichtbar.
- `choose x with` φ `in R`: Die lokale Variable `x` wird angelegt. Der Wert der Variablen `x` wird dabei zufällig gewählt, wobei die Belegung von `x` die Bedingung φ erfüllen muss.
- R_0 `seq` R_1: Die Regeln R_0 und R_1 werden sequentiell nacheinander ausgeführt.

Es existieren noch weitere Regelkonstrukte für ASMs wie etwa die parallele Ausführung von Regeln sowie ein forall-Konstrukt. Da diese aber in der vorliegenden Arbeit nicht verwendet werden, werden sie an dieser Stelle nicht weiter beschrieben.

8.1.5. Das Beweissystem KIV

KIV [9,160,161] ist ein interaktives Beweissystem, das den oben beschriebenen Sequenzenkalkül verwendet. In diesen wurden neben den Regeln für Prädikatenlogik auch entsprechende Regeln für den Beweis von Formeln unter anderem der Logik höherer Ordnung, dynamische Logik, Temporallogik [10] sowie Statecharts [183] integriert. Außerdem sind in KIV Kalküle für die Verifikation von Java-Programmen [180] sowie von QVTO-Modelltransformationen [181] implementiert.

Um die Zahl der manuellen Beweisschritte zu minimieren, kann der Anwender sogenannte Simplifikationsregeln spezifizieren. Diese Theoreme werden von KIV ggf. automatisch auf Be-

weisziele angewendet, um diese zu vereinfachen. Besonderes Augenmerk bei der Entwicklung von KIV wurde auf die hohe Effizienz der Anwendung von Simplifikationsregeln gelegt, so dass das System mehrere tausend Regeln gleichzeitig verwalten kann. Durch geschickte Definition von Simplifikationsregeln kann die Beweisführung stark automatisiert werden.

Die interaktive Anwendung von Beweisregeln wird in KIV in einer graphischen Oberfläche durchgeführt. Diese zeigt neben der aktuellen Beweisverpflichtung auch eine Übersicht über den Beweisbaum des bisherigen Beweises. Ein Wechsel zwischen unterschiedlichen Zielen in diesem Beweisbaum ist ebenso möglich wie die kontextabhängige Anwendung von Beweisregeln auf einzelne Teile der Sequenz. Das Korrektheitsmanagement des Beweissystems unterstützt das automatische Wiederholen von Beweisen, so dass die bereits durchgeführten Beweise bei Änderungen an der Spezifikation weiterhin gültig bleiben, sofern sie nicht von den gemachten Änderungen betroffen sind. Außerdem stellt KIV eine große Standardbibliothek[1] für die Verwendung von Datentypen wie Integern, natürliche Zahlen, Listen oder Mengen zur Verfügung, die die Grundlage für das in dieser Arbeit vorgestellte formale Modell bilden.

ASMs werden in KIV in dynamischer Logik spezifiziert. Eine ASM wie sie z.B. in [25] spezifiziert ist, besteht aus einer Regel, die iteriert angewendet wird. Im KIV-System dagegen sind ASM-Regeln als Prozeduren spezifiziert. Die ASM selber ist eine Prozedur namens `ASM#`, die sich auf eine Prozedur namens `RULE#` abstützt. Letztere enthält die eigentlichen ASM-Zustandsübergangsregeln. Ein Prädikat `final`, welches auf dem globalen Systemzustand definiert ist, regelt die Terminierung der ASM:

```
ASM#(input; state) {while (not final(state)) RULE#(input; state)}
```

`input` ist eine optionale Eingabe an das System, die in dieser Arbeit nicht benötigt wird.

Die eigentlichen Konstrukte der ASMs werden mit der Abbildungsfunktion $[\![]\!]$ wie folgt auf Konstrukte der dynamischen Logik abgebildet:

- $[\![\text{f(p)} := \text{v}]\!] = f(p) := v$
- $[\![\text{if } \varphi \text{ then } R_0 \text{ else } R_1]\!] = \text{if } \varphi \text{ then } [\![R_0]\!] \text{ else } [\![R_1]\!]$
- $[\![\text{let x=t in } R]\!] = \text{var x=t in } [\![R]\!]$
- $[\![\text{choose x with } \varphi \text{ in } R]\!] = \text{choose x with } \varphi \text{ in } [\![R]\!]$
- R_0 seq R_1 wird abgebildet auf $[\![R_0]\!];[\![R_1]\!]$

8.1.6. Theorien zur Verfeinerung

Verfeinerung (Refinement) ist eine Technik zum Korrektheitsnachweis von Systemen. Es gibt verschiedene Verfeinerungstheorien (z.B. [24,46,168]). Für diese Arbeit sind insbesondere zwei dieser Theorien relevant: Data Refinement [83] und ASM Refinement für Invarianten [166], die im Folgenden kurz vorgestellt werden.

Data Refinement:
Ziel des Data Refinements ist es nachzuweisen, dass die Abläufe zweier Systeme in einer Inklusionsbeziehung zueinander stehen, obwohl diese auf Datentypen unterschiedlicher Abstraktionsebenen operieren.

[1] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/

Dazu wird eine Relation `R` zwischen den Zuständen der beiden Systeme definiert und bewiesen, dass für jeden Schritt, den das konkrete System durchführen kann, ein Schritt des abstrakten Systems existiert. Dabei müssen die Zielzustände nach diesen Schritten wieder in Relation zueinander stehen.

Weiterhin muss gezeigt werden, dass zu jedem Startzustand `cs` des konkreten Systems, der mit dem globalen Startzustand `gs` korrespondiert, ein Startzustand `as` des abstrakten Systems existiert, der in Relation `R(as, cs)` steht. Kann dann noch eine entsprechende Korrespondenz der Zielzustände gezeigt werden, ist die Verfeinerung bewiesen.

ASM Refinement:
Es gibt Situationen, in denen die Verfeinerung durch Data Refinement zu eng gefasst ist, da das Data Refinement alle Abläufe des abstrakten Systems im Konkreten erhält. Manchmal ist es jedoch gerade wünschenswert, dass das konkrete System nicht alle Abläufe des abstrakten Systems zulässt. In diesen Fällen kann auf die Technik des ASM-Refinements zurückgegriffen werden. Es gilt, dass eine Verfeinerung, die nach der Data Refinement-Theorie korrekt ist, auch korrekt im Sinne des ASM-Refinements für Invarianten ist.

Beim ASM Refinement dürfen - vereinfachend gesagt - einzelne Zustände des abstrakten Systems als nicht-beobachtbar betrachtet werden. In der Folge ist es nicht mehr notwendig, Eigenschaften über diesen Zuständen zu zeigen.

An die Stelle der starren 1:1 Abbildung des Data Refinements treten dadurch flexiblere m:0..1 Abbildungen, bei denen ein Schritt des konkreten Systems seine Entsprechung in keinem, einem oder beliebig (aber endlich) vielen Schritten des abstrakten Systems findet.

Die Refinement-Relation `R` wird dabei durch den Invarianten-Begriff abgelöst und es werden Bedingungen angegeben, unter denen Invarianten der beobachtbaren Zustände des abstrakten Systems auf die Zustände des konkreten Systems übertragen werden.

Bei der Verifikation von (Sicherheits-)Eigenschaften möchte man, dass sich die über dem abstrakten System bewiesene Aussagen auf die konkreteren, verfeinerten Systeme übertragen. Das in [64] von Grandy sowie in dieser Arbeit verwendete ASM-Refinement für Invarianten erhält die auf abstrakter Ebene bewiesenen (Sicherheits-)Eigenschaften (siehe Kapitel 10), da es sich bei diesen um Invarianten handelt (siehe Kapitel 9).

8.2. Allgemeines zum formalen Modell

Das generierte formale Modell ist eine abstrakte Nachbildung der realen Welt, d.h. die formale Spezifikation umfasst alle beteiligten Kommunikationspartner und ihr Verhalten, die in der Realität möglichen Kommunikationsabläufe sowie das Eingreifen des Angreifers in die Kommunikation. Somit enthält das formale Modell, neben der Spezifikation der Terminal- und Smart Card-Komponenten, auch die Spezifikation der Benutzer und des Angreifers, die zwar im plattformunabhängigen UML-Modell modelliert, in der Realität aber Personen sind und deshalb nicht in Quellcode übersetzt werden (siehe Kapitel 6).

Das formale Modell lässt sich in einen statischen und einen dynamischen Teil untergliedern. Der statische Teil der Anwendung umfasst die Komponenten und Datentypen, die Kommunikationsstruktur sowie das Angreifermodell. Diese Aspekte werden aus den Klassen- und

Deploymentdiagrammen generiert. Der dynamische Teil der Anwendung, d.h. die mit Aktivitätsdiagrammen modellierten Protokolle, wird in eine Abstract State Machine übersetzt.

Im Folgenden werden diese vier Aspekte, d.h. die Komponenten und Datentypen, die Kommunikationsstruktur, der Angreifer sowie die Protokolle, erläutert. Die genauen Regeln, nach denen diese vier Teile generiert werden, sind in Abschnitt 8.4 beschrieben.

Komponenten und Datentypen

Die an einer Anwendungen beteiligten Akteure werden in der formalen Spezifikation als *Agenten* bezeichnet. Diese bestehen aus den im UML-Modell definierten Benutzern, den Smart Card- und Terminalkomponenten sowie dem Angreifer. In der realen Welt werden in den meisten Anwendungen viele Smart Cards und auch mehrere Terminals verwendet. Demnach kann ein Angreifer für seine Angriffe auch mehrere Smart Cards benutzen und diese in verschiedenen Terminals verwenden. Um die Realität nachzubilden, muss der Angreifer diese Möglichkeiten auch im formalen Modell besitzen. Anderenfalls gäbe es eventuell in der Realität Angriffe, die der im formalen Modell spezifizierte Angreifer nicht durchführen kann und die deshalb bei der Verifikation unentdeckt blieben. Von jedem Agententyp, außer dem Angreifer, gibt es deshalb im formalen Modell beliebig, aber endlich viele Instanzen. Die einzelnen Instanzen können anhand eines eindeutigen Namens voneinander unterschieden werden.

Die im UML-Modell definierten Daten- und Nachrichtenklassen sowie die vordefinierten Sicherheitsdatentypen werden ebenfalls in algebraische Spezifikationen übersetzt. Dabei wird für jede UML-Klasse eine gleichnamige Datenspezifikation (`data specification`) erzeugt. Dies ist ein wichtiger Unterschied zu dem formalen Modell des Prosecco-Ansatzes, in dem ein generischer und rekursiver, aber anwendungsunabhängiger Datentyp `Document` für alle Daten- und Nachrichtentypen verwendet wird. Der Vorteil der direkten Übersetzung jeder UML-Klasse in einen anwendungsabhängigen abstrakten Datentyp ist, dass sich das UML-Modell und die generierte formale Spezifikation auf diese Weise sehr ähnlich sind und die Verifikation so erleichtert wird.

An dieser Stelle zeigt sich, dass die Nullfreiheit der Sprache MEL sowie die Sharing- und Zyklenfreiheit der UML-Klassen ein Vorteil bei der Abbildung der UML-Nachrichten- und Datenklassen in abstrakte Datentypen ist. Diese Einschränkungen vereinfachen die Abbildung deutlich.

Abbildung 8.1 zeigt zwei Nachrichtenklassen aus dem plattformunabhängigen UML-Modell der Kopierkartenanwendung. Im Folgenden ist der entsprechende Ausschnitt aus der Datenspezifikation `message` der Kopierkartenanwendung, die alle Nachrichtenklassen für diese Anwendung definiert, abgebildet.

```
data message =
mkLoad(. .amount : int;. .authterminal : HashedData) with isLoad |
mkRequestBalance with isRequestBalance |
..
```

Die Spezifikation enthält einen Konstruktor für jede Nachrichtenklasse, mit dem entsprechende Elemente des Datentyps erzeugt werden können. Der Konstruktor besitzt einen Parameter für jedes Attribut und Assoziationsende der Nachrichtenklasse.

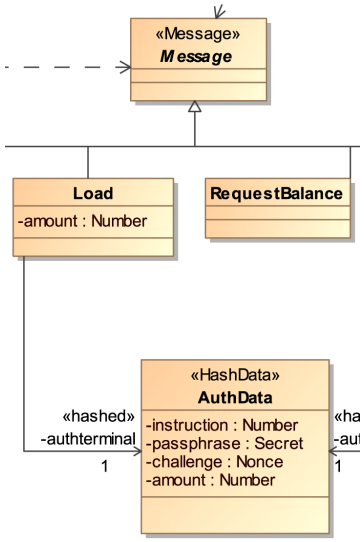

Abbildung 8.1.: Ausschnitt aus dem UML-Anwendungsmodell: Nachrichtenklassen `Load` und `RequestBalance`

Außerdem wird eine Spezifikation generiert, die algebraische Datentypen für die in UML modellierten Datenklassen enthält. Für jede UML-Datenklasse wird ein gleichnamiger algebraischer Datentyp generiert.

Alle Datenklassen werden zusätzlich in einem Datentyp namens `data` zusammengefasst. Dieser ist ein Container für alle abstrakten Datentypen, die aus einer UML-Datenklasse erzeugt wurden.

Die Stereotypen ≪HashData≫, ≪SignData≫ und ≪PlainData≫ werden auf gleichnamige algebraische Datentypen abgebildet. Diese sind ebenfalls frei erzeugt und sind Container für alle Datenklassen, die im UML-Modell mit dem entsprechenden Stereotyp annotiert sind. Für die in UML definierten Sicherheitsdatentypen `HashedData`, `SignedData`, `EncDataSymm`, `EncDataAsymm` und `MAC` werden ebenfalls algebraische Datentypen generiert. Für die Kopierkartenanwendung ist dies nur der Typ `HashedData`, der aus einem Element vom Typ `HashData` erzeugt wird. Hier wird deutlich, dass nur für diejenigen Stereotypen und Sicherheitsdatentypen algebraische Datentypen generiert werden, die bei der Modellierung der konkreten Anwendung verwendet werden.

Kommunikationsstruktur

In der realen Welt können die Agenten einer Anwendung nicht beliebig miteinander kommunizieren. Ein Benutzer kann beispielsweise nicht mit einer Smart Card direkt Nachrichten austauschen, sondern muss dies indirekt über ein Terminal tun, in dessen Kartenleser sich die Smart Card befindet. Diese Einschränkungen an die Kommunikationsstruktur sind auch im formalen Modell abgebildet. Im UML-Modell ist durch das Deploymentdiagramm festgelegt, welche Agenten miteinander kommunizieren können. Abbildung 8.2 zeigt die aus einem De-

ployментdiagramm (siehe Abschnitt 4.2.2) generierte Kommunikationsstruktur im formalen Modell.

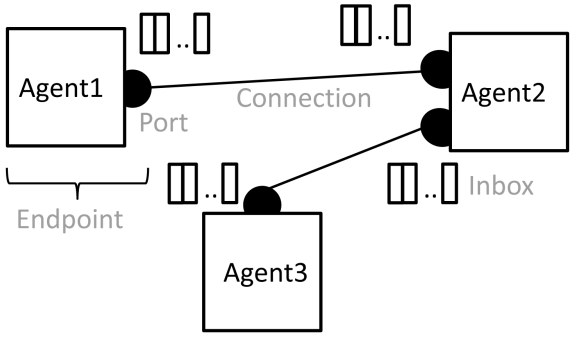

Abbildung 8.2.: Kommunikationsstruktur im formalen Modell

Eine *Connection* (oder Verbindung) besteht aus zwei Endpunkten, wobei ein *Endpunkt* durch einen Agenten und einen *Port* definiert ist. Wie auch im UML-Modell entspricht der Port einem eindeutigen Namen. Die Angabe eines Ports ist notwendig, da in einigen Anwendungen mehr als ein Kommunikationspfad zwischen zwei Agenten existiert. Diese können anhand der Ports voneinander unterschieden werden. Für Ports, die im Deploymentdiagramm nicht benannt sind (siehe Abschnitt 4.2.2.1), werden eindeutige Namen generiert. Jedem Port eines Agenten ist eine Inbox assoziiert, die die Nachrichten enthält, die an den zugehörigen Agenten versendet, aber noch nicht von diesem empfangen und verarbeitet wurden. Die Inbox hat den Namen `inputs` und ist als Liste realisiert.

Um eine Nachricht an einen Agenten senden zu können, muss eine Verbindung zwischen dem sendenden Agenten und dem Empfänger aufgebaut sein. Die existierenden Verbindungen sind in einer Menge aus Verbindungen mit dem Namen `connections` gespeichert. So wird verhindert, dass zu einem Zeitpunkt unterschiedliche Agenten Nachrichten an denselben Port schicken. Dies ist in der realen Welt nicht möglich (z.B. können sich keine zwei Karten zur selben Zeit in einem Kartenlesegerät befinden oder zwei unterschiedliche Benutzer mit einem Terminal, d.h. derselben Benutzeroberfläche, kommunizieren) und wird deshalb auch im formalen Modell unterbunden. Das Senden einer Nachricht kann nur über Verbindungen geschehen, die zum Zeitpunkt des Versendens in der Menge `connections` vorhanden sind. Das Senden einer Nachricht entspricht im formalen Modell dem Schreiben dieser Nachricht in die entsprechende Inbox des Empfängers. Der Empfang einer Nachricht entspricht der Entnahme dieser aus der Inbox. Da der Angreifer, sofern er entsprechende Fähigkeiten hat, ebenfalls Nachrichten in die Inboxen schreiben kann, können sich in einer Inbox theoretisch beliebig viele Nachrichten befinden.

Angreifer

Ein wichtiger Teil des formalen Modells ist der Angreifer. Die modellierte Anwendung wird im SecureMDD-Ansatz als „Black Box“ betrachtet. Dies bedeutet, der Angreifer hat keine Möglichkeit, auf die Agenten selber zuzugreifen, ihren Speicher auszulesen oder zu manipulieren (siehe Abschnitt 2.2, Seite 15). Die Protokollschritte, die von den Smart Card- und

Terminalkomponenten durchgeführt werden, werden somit als atomar betrachtet. Der Angreifer hat keine Möglichkeit, einen Protokollschritt während der Ausführung abzubrechen. Er kann jedoch die Kommunikation zwischen den Agenten manipulieren, (geheime) Informationen abhören und Nachrichten unterdrücken. In der generierten Abstract State Machine, die die Protokolle und Protokollschritte spezifiziert, entspricht die Anwendung einer ASM-Regel einem Protokollschritt. Der ASM-Schritt ist ebenfalls atomar, was zu der „Black Box"-Sicht auf die Anwendung passt. Das Mitlesen, Senden und Unterdrücken von Nachrichten ist im formalen Modell dadurch realisiert, dass der Angreifer die Inboxen der (entsprechenden) Agenten manipulieren kann. Das formale Modell verwendet ein asynchrones Kommunikationsmodell, d.h. das Schreiben einer Nachricht in eine Inbox und die Entnahme dieser Nachricht sind entkoppelt. Auf Codeebene ist die Kommunikation jedoch synchron implementiert. Dies hat zur Folge, dass es im formalen Modell eventuell mehr Abläufe gibt als in der Implementierung. Die Abbildung des Angreifers ist jedoch adäquat zu den Angreiferfähigkeiten in der realen Welt.

Im UML-Modell ist der Angreifer nur implizit modelliert, indem die Kommunikationskanäle mit Angreiferfähigkeiten annotiert werden. Im formalen Modell dagegen ist er explizit als Agent spezifiziert. Im Gegensatz zu den weiteren Agenten gibt es von dem Angreifer nur eine Instanz. Natürlich können in der Realität mehrere Angreifer versuchen mit unterschiedlichen Strategien Schaden anzurichten. Im formalen Modell sind diese zu einem Agenten zusammengefasst, der das Wissen aller Angreifer vereint.

Die formale Verifikation im SecureMDD-Ansatz basiert auf der Annahme der *perfekten Kryptographie*. Dies bedeutet, dass ein Angreifer die verwendeten Algorithmen nicht brechen, d.h. ihre Schutzfunktion nicht umgehen kann. Konkret heißt das, dass es nicht möglich ist, verschlüsselte Daten ohne Kenntnis des Schlüssels zu entschlüsseln, Hashfunktionen zu invertieren und digitale Signaturen zu fälschen. Die Annahme der perfekten Kryptographie bedingt auch, dass alle Operationen injektiv und somit z.B. alle Hashfunktionen kollisionsfrei sind. In der Praxis gilt die perfekte Kryptographie natürlich nicht. Es werden z.B. immer wieder Angriffe bekannt, bei denen Kollisionen für Hashfunktionen gefunden werden (z.B. [107]). Eine weitere Annahme der perfekten Kryptographie ist, dass ein Angreifer die geheimen Passwörter der Nutzer der zu verifizierenden Anwendung nicht erraten kann. Die Tatsache, dass in der Praxis viele unsichere und leicht zu erratende Passwörter verwendet werden bzw. geheime Daten durch Phishing-Angriffe in Erfahrung gebracht werden können, wird im formalen Modell nicht betrachtet. Die Annahme der perfekten Kryptographie ist im Bereich der Verifikation von kryptographischen Protokollen üblich. Auch andere wichtige Arbeiten auf diesem Gebiet [12, 34, 117, 152] treffen diese Annahme. Ohne sie ist es in der Regel nicht möglich, die Sicherheit eines Protokolls formal nachzuweisen. Geht man davon aus, dass ein Angreifer einen geheimen Schlüssel durch einen Brute-Force Angriff lernen kann, muss man davon ausgehen, dass er alle mit diesem Schlüssel verschlüsselten Klartextdaten ebenfalls erfährt. Unter dieser Annahme werden die allermeisten Protokolle nicht mehr als sicher nachzuweisen sein.

Das formale Modell speichert explizit das Wissen des Angreifers, d.h. alle Informationen, zum Beispiel Nonces, Geheimnisse oder kryptographische Schlüssel, die dem Angreifer bekannt sind. Sein Wissen kann der Angreifer verwenden, um z.B. ein verschlüsseltes Dokument, das er beim Versand an einen Agenten mitliest, zu entschlüsseln (sofern der entsprechende Schlüssel in seinem Wissen ist) oder Nachrichten zu erzeugen bzw. zu manipulieren. Nichtkryptographische Daten wie Zahlen oder Strings kann der Angreifer immer erzeugen. Diese

müssen somit nicht explizit im Angreiferwissen gespeichert werden. Bei den kryptographischen Daten ist es jedoch wichtig, dass der Angreifer nur diejenigen kennt, die er initial in seinem Wissen hat (z.B. kann der Angreifer gleichzeitig ein legaler Benutzer der Anwendung sein und somit seine eigene PIN-Nummer kennen) oder die er während der Laufzeit der Anwendung beim Mitlesen der Kommunikation erfährt. Anderenfalls wäre es möglich, dass der Angreifer z.B. immer den richtigen, ihm allerdings in der Praxis unbekannten, Schlüssel verwendet, und auf diese Weise ein verschlüsseltes Dokument entschlüsselt. Unter dieser Annahme wäre die Anwendung unsicher und vermutlich keine Sicherheitsaussage beweisbar.

Das dynamische Verhalten: Die Protokolle

Das dynamische Verhalten der Anwendung ist im formalen Modell durch eine Abstract State Machine (ASM) beschrieben. Die Kommunikation zwischen den Agenten ist dabei asynchron realisiert. Eine Komponente wartet nicht auf den Empfang einer Nachricht, sondern es wird nichtdeterministisch ausgewählt, welche Komponente (mit nichtleerer Inbox) eine Nachricht empfangen und einen Schritt ausführen soll. Anders als in der Implementierung (bei der z.B. ein Terminal synchron auf den Empfang der Antwortnachricht von der Smart Card wartet), gibt es somit im formalen Modell keine erzwungene feste Reihenfolge zwischen den Protokollschritten. Die auf diese Weise durch die ASM abgebildeten Abläufe sind eine Obermenge der in der Realität möglichen (endlichen) Abläufe. Insbesondere sind alle Protokollabläufe möglich, die auch in der realen Welt durchführbar sind.

Die ASM ist als Prozedur realisiert, die im nachfolgenden Listing dargestellt ist.

```
ASM(...) {stop := false; while(¬ stop) {STEP; stop := [?];}}
```

Die ASM ruft die Regel STEP auf, die einen ASM-Schritt durchführt. Anschließend wird das boolesche Flag `stop` nichtdeterministisch auf `true` oder `false` gesetzt. Wird der Wert `false` gewählt, findet kein weiterer ASM-Schritt statt. Anderenfalls wird wieder nichtdeterministisch der nächste ASM-Schritt ausgewählt. Dadurch werden alle endlichen Präfixe aller Traces erzeugt.

```
1  let asm-step = [?] in {
2    if asm-step = attacker-agent-step
3    then {ATTACKER(..)}
4    else if asm-step = user-agent-step
5         then {choose ag with (is_user(ag)) in USER(..)}
6         else if asm-step = Copycard-agent-step
7              then {choose ag with
8                      is_Copycard(ag) in COPYCARDSTEP(..)}
9              else ..
10 };
```

Das Listing zeigt (einen Teil) der ASM-Regel STEP der Kopierkartenanwendung, die im KIV-System ebenfalls als Prozedur definiert ist. Die ASM-Regel wählt nichtdeterministisch aus, welcher Agententyp einen Schritt durchführen soll (Zeile 1). Möglich sind ein Schritt des

Angreifers, einer Kopierkarte, eines Ladeterminals oder eines Kopiergeräts. Außerdem gibt es noch einen Schritt, der (nichtdeterministisch) eine neue Verbindung aufbaut und einen, der eine (nichtdeterministisch gewählte) bestehende Verbindung abbaut. Wird beispielsweise ein Schritt auf der Kopierkarte ausgewählt (Zeile 6), wird zunächst ein konkreter Agent vom Typ `Copycard` ausgewählt, der einen Schritt durchführen soll, und für diesen die ASM-Regel COPYCARDSTEP aufgerufen (Zeile 7-8). Die Schritte für die Terminals und Benutzer sind analog spezifiziert.

Die ASM-Regel COPYCARDSTEP ist ebenfalls als Prozedur spezifiziert. Ein Ausschnitt hieraus ist nachfolgend abgebildet.

```
1  COPYCARDSTEP {
2  choose port with validport(ag, port) ∧ inputs(ag)(port)≠[] in {
3    let inmsg = inputs(ag)(port).first in {
4        inputs := rem(ag, port, inputs);
5        if (isLoad(inmsg)) then {Load} else
6        ..
7        else STOPSTEP}
8     }
9  ifnone skip
10 };
```

Die Regel COPYCARDSTEP wählt zunächst nichtdeterministisch einen Port des gewählten Agenten aus, dessen Inbox nicht leer ist (Zeile 2). Dann wird die erste Nachricht aus der Inbox entnommen und dort gelöscht (Zeilen 3-4) und je nach Nachrichtentyp eine Unterregel aufgerufen, die diese Nachricht verarbeitet und den entsprechenden Protokollschritt für den gewählten Agenten durchführt (Zeile 5 für eine Nachricht vom Typ `Load`). Nach Durchführung des Protokollschritts ist der ASM-Schritt abgeschlossen. Existiert kein Port mit nichtleerer Inbox, ist der Schritt beendet (Zeile 9).

Wird der Angreifer für die Durchführung eines Schrittes ausgewählt, wird nichtdeterministisch ausgewählt, ob der Angreifer eine Nachricht senden, Nachrichten unterdrücken oder eine Verbindung abbauen soll. Im ersten Fall wird nichtdeterministisch eine Verbindung gewählt, über die der Angreifer Nachrichten senden kann und anschließend aus dem Wissen des Angreifers eine Nachricht abgeleitet, die in die Inbox des Empfängers geschrieben wird. Im zweiten und dritten Fall wird nichtdeterministisch eine Inbox geleert bzw. eine existierende Verbindung abgebaut. Das Lesen einer gesendeten Nachricht geschieht nicht durch einen expliziten ASM-Schritt, sondern beim Senden einer Nachricht über eine Verbindung, die der Angreifer abhören kann.

Die im formalen Modell möglichen Abläufe (für Terminal, Karten und Benutzeragenten) sind somit wie folgt spezifiziert:

1. Wähle nichtdeterministisch, ob ein (weiterer) ASM-Schritt gemacht werden soll. Falls nein: Beende die Ausführung der ASM, falls ja: siehe 2.
2. Wähle nichtdeterministisch, welcher Agententyp einen Schritt durchführen soll
3. Wähle nichtdeterministisch, welcher konkrete Agent des gewählten Typs einen Schritt durchführen soll

4. Wähle nichtdeterministisch, aus welcher (nichtleeren) Inbox des gewählten Agenten eine Nachricht entnommen wird
5. Verarbeite diese Nachricht und führe den entsprechenden Protokollschritt aus
6. gehe zurück zu 1.

Das formale Modell berücksichtigt, dass es beliebig (aber endlich) viele Instanzen der verschiedenen Agententypen (d.h. Kopierkarten, Ladeterminals, Kopiergeräte und Kartenbesitzer bei der Kopierkartenanwendung) gibt und dass verschiedene Protokollläufe auch verschachtelt (d.h. interleaved) ablaufen können. Jeder Protokolllauf kann von dem Angreifer (unter Verwendung seiner Einflussmöglichkeiten auf das System, die in den Deploymentdiagrammen beschrieben sind), manipuliert werden.

8.3. Der Prosecco-Ansatz

Das in Abschnitt 8.2 vorgestellte formale Modell ist eine Weiterentwicklung des Prosecco-Ansatzes von Haneberg [78]. Der Prosecco-Ansatz ermöglicht die formale Verifikation von Smart Card-Anwendungen. Er basiert ebenfalls auf algebraischen Spezifikationen und Abstract State Machines und verwendet die gleichen Konzepte (Connections, Spezifikation des Angreifers, Aufbau der ASM) wie das formale Modell des SecureMDD-Ansatzes.

Die Idee des Prosecco-Ansatzes ist es jedoch, die Spezifikation der Daten, Nachrichten sowie des Angreiferwissens möglichst unabhängig von der konkreten Anwendung zu machen. Der Prosecco-Ansatz verwendet hierfür einen generischen, rekursiv definierten Datentyp namens `Document`, mit dem sich alle Daten- und Nachrichtenklassen darstellen lassen.

```
data Document =
    IntDoc(value : int) with isIntDoc
  | NonceDoc(nonce : Nonce) with isNonceDoc
  | SecretDoc(secret : Secret) with isSecretDoc
  | KeyDoc(key : Key) with isKeyDoc
  | HashDoc(hash : Document) with isHashDoc
  | EncDoc(key : Key, doc : Document) with isEncDoc
  | SigDoc(key : Key, doc : Document) with isSigDoc
  | Doclist(docs : Documentlist) with isDoclist

data Documentlist =
    [] with isEmptyList
  | .+. (.first : Document, .rest : Documentlist)
                                  with isDocumentlist
```

Auch das Angreiferwissen ist als eine Menge von `Documents` definiert. Dies hat den Vorteil, dass eingie benötigte Lemmatas ebenfalls anwendungsunabhängig sind und somit über eine Bibliothek zur Verfügung gestellt werden können. Der Nachteil ist, dass durch die Verwendung des geschachtelten und rekursiv definierten Datentyps die Beweise teilweise recht schwierig und komplex werden. Dies gilt insbesondere für die Berechnung des aktuellen Angreiferwis-

sens. Außerdem sind die Spezifikationen und die ASM durch die Verwendung des generischen Datentyps teilweise schwer zu lesen. Dies sind große Nachteile für die interaktive Verifikation.

Im SecureMDD-Ansatz werden die im UML-Modell definierten Daten- und Nachrichtenklassen direkt in entsprechende algebraische Datentypen übersetzt und nicht als `Documents` kodiert. Es hat sich herausgestellt, dass die Daten- und Nachrichtenklassen für die allermeisten Anwendungen nicht tief geschachtelt und nicht rekursiv sind. Aus diesem Grund sind die meisten Lemmatas einfacher zu beweisen als bei Verwendung eines generischen Datentyps. Insgesamt reduzieren sich somit die Komplexität und der Umfang der Beweise.

Ein weiterer Vorteil des modellgetriebenen SecureMDD-Ansatzes ist, dass das generierte formale Modell auf die konkrete Anwendung zugeschnitten ist. So ist das formale Modell einer kleinen Anwendung mit wenigen Datentypen und Protokollschritten deutlich übersichtlicher als das einer großen Anwendung. Sicherheitsdatentypen und kryptographische Operationen, die von der Anwendung nicht benötigt werden, werden nicht in das formale Modell übersetzt. Verwendet die Anwendung z.B. keine Verschlüsselung, ist die Berechnung des Angreiferwissens deutlich einfacher und weniger komplex.

8.4. Generierung der formalen Spezifikation

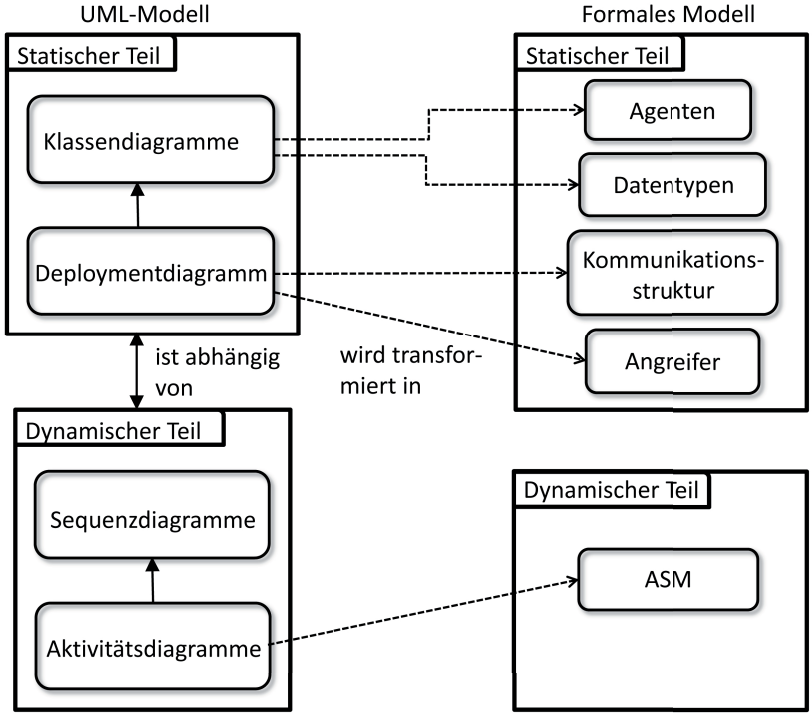

Abbildung 8.3.: Transformation der UML-Diagramme in das formale Modell

In diesem Abschnitt wird die Transformation des plattformunabhängigen UML-Modells in das formale Modell beschrieben. Diese Transformation definiert die Semantik des plattformunab-

hängigen UML-Modells. Die Semantik der Model Extension Language wurde in Abschnitt 5.2 beschrieben.

In Abbildung 8.3 ist dargestellt, aus welchen UML-Diagrammen die verschiedenen Teile der formalen Spezifikation (Agenten, Datentypen, Kommunikationsstruktur, Angreifer und ASM) erzeugt werden.

Die Agenten sowie die algebraischen Datentypen werden aus den UML-Klassendiagrammen generiert. Die Informationen, die für die formale Spezifikation des Angreifers und der Kommunikationsstruktur benötigt werden, werden dem UML-Deploymentdiagramm entnommen. Dieser statische Teil der Anwendung ist in algebraischen Spezifikationen definiert. Der dynamische Teil der Anwendung, beschrieben durch die in den Aktivitätsdiagrammen modellierten Protokolle, ist als Abstract State Machine spezifiziert. Die Sequenzdiagramme enthalten zu den Aktivitätsdiagrammen redundante Informationen und werden deshalb bei der Generierung des formalen Modell nicht berücksichtigt (siehe Abschnitt 4.4, Seite 71).

Im Folgenden werden in Abschnitt 8.4.1 die Transformationsregeln für UML-Klassendiagramme beschrieben. Abschnitt 8.4.2 erläutert die Transformation der Deploymentdiagramme und Abschnitt 8.4.3 die Abbildung der Aktivitätsdiagramme.

8.4.1. Transformation der Klassendiagramme

Die Transformation der Klassendiagramme lässt sich in verschiedene Teilaspekte unterteilen. Diese sind in der nachfolgenden Tabelle aufgelistet und werden im Anschluss an die Tabelle erläutert.

	UML und MEL	formales Modell
1.	primitive Datentypen Number, Boolean, String	abstrakte Datentypen int, bool, string
2.	UML-Klassen für die Sicherheitsdatentypen und vordefinierte kryptographische Operationen in MEL	abstrakte Datentypen und Funktionen bzw. Prozeduren
3.	Zertifikatklasse und vordefinierte MEL-Operationen	abstrakter Datentyp und Funktionen
4.	Komponenten- und Benutzerklassen	abstrakter Datentyp (Agent)
5.	Attribute und (gerichtete) Assoziationsenden der Komponentenklassen	dynamische Funktionen
6.	Datenklassen	abstrakte Datentypen
7.	(Benutzer-)Nachrichtenklassen	abstrakte Datentypen
8.	Stereotypen ≪HashData≫, ≪SignData≫ und ≪PlainData≫	abstrakte Datentypen
9.	Assoziationen mit Multiplizität größer als eins (Listen) und MEL Listenoperationen	Aktualisierung der Listenspezifikation und Funktionen
10.	Zustandsklasse (Status) einer Komponente	abstrakter Datentyp
11.	Klasse mit Stereotyp ≪Constants≫	Konstanten
12.	manuelle Methoden (Annotation einer Operation mit Stereotyp ≪Manual≫)	Funktionen (unspezifiziert)

1. Primitive Datentypen Number, Boolean und String

Der in UML verwendete primitive Typ `Number` wird bei der Generierung des formalen Modells in den Typ `int` übersetzt. Dieser ist in der Basisbibliothek des KIV-Systems definiert und wird in das formale Modell importiert. Der primitive Typ `Boolean` sowie der Typ `String`, die ebenfalls in UML definiert sind, werden in die Typen `bool` und `string` übersetzt. Diese sind ebenfalls Teil der Basisbibliothek und werden in das formale Modell importiert.

2. Sicherheitsdatentypen und kryptographische Operationen

In diesem Abschnitt wird die Transformation der UML-Sicherheitsdatentypen (siehe Abschnitt 4.1.2) sowie der auf diesen Datentypen operierenden kryptographischen MEL-Operationen (siehe Abschnitt 5.4) erläutert.

Nonce:
Die UML-Klasse `Nonce`, die ein Attribut `nonce` vom Typ `String` zur Speicherung der Zufallszahl besitzt, wird in einen abstrakten Datentyp mit Namen `Nonce` übersetzt.

```
data Nonce = mkNonce( . .nonce : string);
```

Die MEL-Operation `generateNonce()`, die eine neue Nonce erzeugt, wird im formalen Modell in eine gleichnamige Prozedur übersetzt. Um sicherzustellen, dass jedes Element nur einmal herausgegeben wird, d.h. eine im formalen Modell neu generierte Nonce fresh ist, basiert die Berechnung einer neuen Nonce auf einem Pool von (duplikatfreien) Nonces. Beim Aufruf der Prozedur wird ein (noch nicht verwendetes) Element aus diesem Pool herausgegeben.

Secret:
Die UML-Klasse `Secret` mit Attribut `secret` vom Typ `String` wird in einen abstrakten Datentyp mit Namen `Secret` übersetzt.

```
data Secret = mkSecret( . .secret : string);
```

Kryptographische Schlüssel:
Die UML-Klassen `SymmKey`, `PublicKey` und `PrivateKey` werden in gleichnamige abstrakte Datentypen übersetzt. Die abstrakte UML-Klasse `Key`, die die gemeinsame Oberklasse der drei konkreten Schlüsselklassen ist, ist im formalen Modell nicht vorhanden.

```
data
SymmKey = mkSymmKey( . .key : string);
PrivateKey = mkPrivateKey( . .key : string);
PublicKey = mkPublicKey( . .key : string);
```

Alle drei Datentypen werden über einen Konstruktor erzeugt, der einen Parameter `key` für den eigentlichen Schlüssel besitzt. Dieser entspricht dem Attribut `key` der UML-Klasse `Key`.

Die MEL-Operation `generateKey()`, die für alle drei Schlüsseltypen definiert ist, wird in eine gleichnamige Prozedur übersetzt. Die Generierung von Schlüsseln basiert im formalen

Modell auf Nonces und es wird derselbe Pool von Nonces verwendet, der auch für das Generieren einer neuen Nonce dient. Ein Schlüssel ist demnach eine zufällige Zahl. Es ist sichergestellt, dass derselbe Schlüssel nicht zweimal generiert wird und Nonces und Schlüssel nicht denselben Wert haben.

HashedData:
Die UML-Klasse `HashedData` wird in einen gleichnamigen abstrakten Datentyp übersetzt.

```
data HashedData = mkHashedData( . .hash : HashData);
hash(hashdata) = mkHashedData(hashdata);
```

Anders als im UML-Modell und in der Implementierung enthält der Datentyp `HashedData` den nicht-gehashten Wert, der aber von dem Angreifer nicht ausgelesen werden kann. Dieser ist über den Selektor `.hash` zugreifbar. Der Typ `HashData` entspricht dem in UML verwendeten Stereotyp ≪HashData≫. Alle Datentypen, über denen ein Hashwert gebildet werden kann (d.h. deren UML-Datenklasse mit ≪HashData≫ annotiert sind), sind im formalen Modell vom Typ `HashData`.

Die MEL-Operation `hash` wird in eine gleichnamige Funktion übersetzt. Diese ruft den Konstruktor des Datentyps `HashedData` auf und erzeugt so ein neues Element.

SignedData:
Die UML-Klasse `SignedData` wird ebenfalls in einen gleichnamigen abstrakten Datentyp übersetzt.

```
data SignedData = mkSignedData( . .key : PrivateKey;
                                . .signdata : SignData);
sign(privkey, signdata) = mkSignedData(privkey, signdata);
verify(pubkey, mkSignedData(privkey, signdata), signdata2) ↔
            is_keypair(pubkey, privkey) ∧ signdata = signdata2;
```

Anders als im UML-Modell und im generierten Quellcode (bei denen nur das signierte Datum gespeichert wird) speichert der Datentyp `SignedData` den privaten Schlüssel, mit dem die Signatur erstellt wurde sowie den Klartext, über dem die Signatur gebildet wurde. Der Typ `SignData` entspricht dem in UML verwendeten Stereotyp ≪SignData≫. Alle Datentypen, über denen eine Signatur gebildet werden kann (d.h. deren UML-Datenklasse mit ≪SignData≫ annotiert sind), sind im formalen Modell vom Typ `SignData`.

Die MEL-Operationen `sign` und `verify` werden in eine gleichnamige Funktion bzw. Prädikat übersetzt. Die Funktion `sign` ruft den Konstruktor des Datentyps `SignedData` auf und erzeugt ein neues Element des Typs. Das Prädikat `verify` ist genau dann wahr, d.h. die Signatur ist gültig, wenn der private und öffentliche Schlüssel ein gültiges Schlüsselpaar sind (`is_keypair`) und die Signatur über dem Klartext erstellt wurde, der als Argument `signdata2` übergeben wurde.

EncDataSymm und EncDataAsymm:
Die UML-Klassen `EncDataSymm` und `EncDataAsymm` werden ebenfalls in gleichnamige abstrakte Datentypen übersetzt. Die abstrakte Oberklasse `EncData` ist im formalen Modell weggelassen.

```
data EncDataSymm
 = mkEncDataSymm( . .key : SymmKey;
                              . .plain : PlainData);
data EncDataAsymm
 = mkEncDataAsymm( . .key : PublicKey; . .plain : PlainData);
```

Beide Datentypen speichern den Schlüssel, mit dem der Klartext verschlüsselt wurde sowie den Klartext. Dies ist ein Unterschied zu dem UML-Modell und der Implementierung, in denen nur die verschlüsselten Daten gespeichert werden.

Die MEL-Operationen `encrypt` und `decrypt`, die sowohl für die symmetrische als auch die asymmetrische Verschlüsselung definiert sind, werden in gleichnamige Funktionen übersetzt.

```
encrypt(symmkey, plaindata) = mkEncDataSymm(symmkey, plaindata);

can_decrypt(symmkey, mkEncDataSymm(symmkey2, plaindata))
   ↔ symmkey = symmkey2;

can_decrypt(symmkey, mkEncDataSymm(symmkey2, plaindata)) →
 decrypt(symmkey, mkEncDataSymm(symmkey2, plaindata)) = plaindata;
```

Die Funktion `encrypt` ruft den Konstruktor des Datentyps `EncDataSymm` auf, der ein neues Element erzeugt. Das Prädikat `can_decrypt` sagt aus, dass ein Element vom Typ `EncDataSymm` entschlüsselt werden kann, wenn der dazu verwendete symmetrische Schlüssel gleich dem Schlüssel ist, mit dem ursprünglich verschlüsselt wurde. Die Funktion `decrypt` gibt das Klartextobjekt zurück, sofern der richtige Schlüssel zum Entschlüsseln verwendet wird. Anderenfalls ist die Funktion unspezifiziert.

Die Funktionen zum asymmetrischen Ver- und Entschlüsseln sind analog definiert.

MACData:
Für die UML-Klasse `MACData` wird analog zu den anderen Sicherheitsdatentypen ein abstrakter Datentyp generiert. Die MEL-Operationen `computeMAC` und `decryptMAC` sind im formalen Modell ebenfalls über Funktionen definiert. Diese stützen sich auf die Funktionen der Datentypen `EncDataSymm` und `HashedData`.

3. Zertifikatklassen

Zertifikate werden in UML über zwei Klassen modelliert. Eine Klasse repräsentiert das Zertifikat (bestehend aus Daten und einer Signatur), die andere enthält die Daten des Zertifikats. Die Modellierung ist in Abschnitt 4.2.1.5 erläutert. Abbildung 8.4 zeigt diese nochmals an einem Beispiel.

Beide UML-Klassen sind im formalen Modell als abstrakte Datentypen spezifiziert.

Die Konstruktoren der Datentypen enthalten einen Parameter für jedes Attribut und Assoziationsende der Klasse. Für die `Zertifikatklasse` sind dies die Assoziationsenden `data` und `sig`, für die Datenklasse die Attribute `name` und `pubkey`.

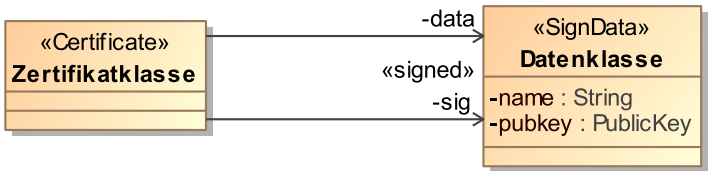

Abbildung 8.4.: Modellierung eines Zertifikats in UML

```
data
Zertifikatklasse = mkZertifikatklasse( . .d : Datenklasse;
                                       . .sig : SignedData);
Datenklasse = mkDatenklasse( . .name : string;
                             . .pubkey : PublicKey);
```

MEL definiert anwendungsabhängig für jede Zertifikatklasse zwei Methoden `generateCertificate` und `verifyCertificate`. Im formalen Modell sind diese über eine gleichnamige Funktion bzw. Prädikat spezifiziert, die sich auf die Methoden `sign` und `verify` des Datentyps `SignedData` stützen.

```
generateCertificate(privkey, datenklasse) =
   mkZertifikatklasse(datenklasse, sign(privkey,datenklasse));
verifyCertificate(pubkey, zertifikatklasse)
↔ verify(pubkey, zertifikatklasse.sig, zertifikatklasse.d);
```

`datenklasse` und `zertifikatklasse` sind Variablennamen. Die Funktion `generateCertificate` ruft den Konstruktor der `Zertifikatklasse` auf. Dieser erhält als Argumente den Wert der Variablen `datenklasse` sowie die Signatur, die mithilfe des Schlüssels `privkey` über der Variablen `datenklasse` bzw. ihrem Wert erzeugt wird. Das Prädikat `verifyCertificate` überprüft die in der Zertifikatklasse gespeicherte Signatur.

Sind im UML-Modell mehrere Zertifikate modelliert, sind die Funktion `generateCertificate` sowie das Prädikat `verifyCertificate` überladen. Für jede modellierte Zertifikatklasse haben die Funktion sowie das Prädikat jedoch unterschiedliche Signaturen, weswegen es zu keinem Konflikt bei der Axiomatisierung kommt.

4. Komponenten- und Benutzerklassen

Die Datenspezifikation `agent` repräsentiert alle Instanzen der im UML-Modell modellierten Smart Card- und Terminalkomponenten und der Benutzer. Außerdem zählt der Angreifer zu den Agenten. Die UML-Komponentenklassen (mit Stereotyp ≪Terminal≫ oder ≪Smartcard≫) sowie die Benutzerklassen (mit Stereotyp ≪User≫) werden im formalen Modell in eine Datenspezifikation mit Namen `agent` übersetzt. Haben mehrere Komponentenklassen eine gemeinsame, abstrakte Oberklasse, wird diese bei der Generierung des formalen Modells entfernt und die konkreten Subklassen erhalten alle Attribute und Assozia-

tionsenden der Oberklasse. Dies entspricht dem Vorgehen bei der Generierung des Quellcodes. Ein `agent` repräsentiert somit alle konkreten Komponentenklassen (`DepositMachine`, `CopyingMachine` und `Copycard` für die Kopierkartenanwendung). Zusätzlich sind alle Benutzerklassen (`CardOwner`) zu einem Benutzer mit Namen `user` zusammengefasst. Für die Kopierkartenanwendung ergibt sich somit folgende Spezifikation:

```
data agent
   = DepositMachine( . .name : nat) with is_DepositMachine |
     CopyingMachine( . .name : nat) with is_CopyingMachine |
     Copycard( . .name : nat) with is_Copycard |
     user( . .name : nat) with is_user |
     attacker with is_attacker
```

Für jeden Agententyp gibt es ein Prädikat (z.B. `is_DepositMachine`), mit dem abgefragt werden kann, ob ein Agent von dem jeweiligen Typ ist. Das formale Modell ermöglicht eine beliebige, aber endliche Anzahl an Instanzen eines Typs. Die verschiedenen Instanzen können anhand eines eindeutigen Namens `name` vom Typ `nat` unterschieden werden. Von dem Angreifer gibt es nur eine Instanz.

Für jeden Agententyp (außer dem Angreifer) wird eine Konstante generiert, die angibt, wie viele Agenten dieses Typs es gibt. Der genaue Wert dieser Konstanten vom Typ `nat` bleibt unspezifiziert, der Wert muss jedoch ungleich null sein.

Im folgenden Listing sind beispielhaft die Konstanten der Kopierkartenanwendung abgebildet.

```
constants
   NUMOFDEPOSITMACHINES : nat;
   NUMOFCOPYINGMACHINES : nat;
   NUMOFCOPYCARDS : nat;
   NUMOFUSERS : nat;

axioms
   NUMOFDEPOSITMACHINES ≠ 0;
   NUMOFCOPYINGMACHINES ≠ 0;
   NUMOFCOPYCARDS ≠ 0;
   NUMOFUSERS ≠ 0;
```

Für die Komponentenklasse `Copycard` zum Beispiel gibt es im formalen Modell die konkreten Agenten `Copycard(1)`, `Copycard(2)`, ... `Copycard(NUMOFCOPYCARDS)`.

5. Attribute und Assoziationsenden der Komponentenklassen

Die Attribute und Assoziationsenden der Komponentenklassen werden in dynamische Funktionen übersetzt. Für jedes Attribut und Assoziationsende wird eine dynamische Funktion generiert. Diese erhält als Eingabe einen Agenten (vom Typ `agent`) und gibt den Wert des entsprechenden Attributs bzw. Assoziationsendes dieses Agenten zurück. Der Typ des Ausgabeparameters entspricht dem (übersetzten) Typ des Attributs bzw. Assoziationsendes. Um die

dynamischen Funktionen besser unterscheiden und bei der Verifikation zuordnen zu können, wird jeweils der Name der UML-Klasse, in der das Attribut (bzw. Assoziationsende) definiert ist, vorangestellt. Dies dient einer besseren Lesbarkeit und erlaubt gleiche Attributnamen für unterschiedliche Komponentenklassen.

Die dynamischen Funktionen der Kopierkartenanwendung sind in Listing 8.1 dargestellt.

```
Copycard-balance : agent → int;
Copycard-challenge : agent → Nonce;
Copycard-passphrase : agent → Secret;
Copycard-statecard : agent → StateCard;
CopyingMachine-amountToPay : agent → int;
CopyingMachine-challenge : agent → Nonce;
DepositMachine-amountToLoad, Terminal-money : agent → int;
Terminal-passphrase : agent → Secret;
Terminal-stateterminal : agent → StateTerminal;
```

Listing 8.1: Dynamische Funktionen der Kopierkartenanwendung

Die dynamischen Funktionen werden von der Abstract State Machine verwendet, die den durch diese Funktionen definierten Zustand der (ausgewählten) Komponente während eines Protokollschritts ausliest und aktualisiert (siehe Abschnitt 8.4.3).

Die Benutzerklassen können theoretisch ebenfalls Attribute und Assoziationen besitzen. Da für einen Benutzer jedoch keine eigene Protokolllogik definiert ist, sondern dieser lediglich Eingaben an das System macht (und auf diese Weise Protokollläufe startet) und Antwortnachrichten empfängt, werden diese Attribute und Assoziationsenden bei der Generierung des formalen Modells ignoriert.

Der im UML-Modell verwendete Stereotyp ≪Initialize≫ wird bei der Generierung der formalen Spezifikation ebenfalls ignoriert. Er ist nur für die Codegenerierung relevant, da dort ein spezieller Initialisierungsmechanismus (für die Smart Cards) bzw. ein Konstruktor mit entsprechenden Parametern (für die Terminals) generiert wird.

Prinzipiell liegt die Initialisierung eines Systems, d.h. die Angabe der konkreten Initialwerte, sowohl für den generierten Quellcode als auch das formale Modell außerhalb des SecureMDD-Ansatzes. Dies bedeutet, dass vor Inbetriebnahme einer Anwendung die Karten und Terminals entsprechend manuell (über den automatisch generierten Mechanismus) initialisiert werden müssen. Dasselbe gilt für das formale Modell, in dem vor der Verifikation manuell angegeben werden muss, mit welchen Werten die dynamischen Funktionen (und das Angreiferwissen) initialisiert werden sollen. Der Grund hierfür ist, dass es zum Teil schwierig ist, die Initialisierungsbedingungen im UML-Modell zu definieren. Beispielsweise ist es bei der Kopierkartenanwendung egal, welche konkrete `passphrase` die Komponenten als gemeinsames Geheimnis verwenden. Es ist jedoch wichtig, dass dieses Geheimnis bei allen Kopierkarten sowie allen Terminals dasselbe ist und der Angreifer es nicht kennt.

6. Datenklassen

Alle UML-Datenklassen (siehe Abschnitt 4.2.1.2) werden in einen abstrakten Datentyp gleichen Namens übersetzt. Ein Element dieses Typs kann über einen Konstruktor erzeugt werden, der Parameter für alle Attribute und Assoziationsenden der Datenklasse besitzt. Ist im UML-Diagramm für eine Datenklasse ein Konstruktor definiert, entspricht die Reihenfolge der Parameter der im Konstruktor festgelegten Reihenfolge.

Dies ist beispielhaft für die Datenklasse `AuthData` der Kopierkartenanwendung dargestellt.

```
data AuthData
 = mkAuthData( . .instruction : int;  . .passphrase : Secret;
               . .challenge : Nonce;  . .amount : int);

data dataclasses
 = wrapAuthData2data( . .authData : AuthData);
```

Zusätzlich wird eine Datenspezifikation mit Namen `dataclasses` generiert. Diese fasst alle UML-Datenklassen der Anwendung zusammen. Sie enthält einen Konstruktor (bzw. Wrapper) für jede Datenklasse, mit dem ein Element des jeweiligen Typs erzeugt werden kann. In der Kopierkartenanwendung gibt es nur die Datenklasse `AuthData`.

Für jedes Attribut und Assoziationsende einer Datenklasse wird zusätzlich eine Updatefunktion definiert, mit der auf das Attribut bzw. Assoziationsende zugegriffen werden kann. Folgendes Listing zeigt beispielhaft die für die Klasse `AuthData` generierten Updatefunktionen.

```
functions
. .instruction:= . : AuthData × int → AuthData;
. .passphrase:= . : AuthData × Secret → AuthData;
. .challenge:= . : AuthData × Nonce → AuthData;
. .amount:= . : AuthData × int → AuthData;

axioms
   authdata.instruction:= i
 = mkAuthData(i, authdata.passphrase, authdata.challenge,
                                          authdata.amount);
   authdata.passphrase:= secret
 = mkAuthData(authdata.instruction, secret, authdata.challenge,
                                             authdata.amount);
...
```

Zum Beispiel definiert die Funktion `.instruction` den Zugriff auf das Attribut `instruction` des Datentyps `AuthData`. `authdata`, `i` und `secret` sind Variablen (vom Typ `AuthData`, `int` bzw. `Secret`).

7. Nachrichtenklassen

Benutzernachrichtenklassen und Nachrichtenklassen (d.h. Klassen, die mit Stereotyp ≪Usermessage≫ bzw. ≪Message≫ annotiert sind) werden im formalen Modell gleichbehandelt und zu einem abstrakten Datentyp `message` zusammengefasst. Dies hat den Vorteil, dass es nur einen Typ von Inboxen gibt. Für die (Benutzer-)Nachrichtenklassen wird deshalb eine Datenspezifikation `message` generiert. Diese fasst alle (Benutzer-)Nachrichtenklassen zusammen, d.h. enthält einen Konstruktor für jede (Benutzer-)Nachrichtenklasse, die im UML-Modell definiert ist. Ist im UML-Modell ein Konstruktor für die Klasse definiert, entspricht die Reihenfolge der Parameter der im UML-Konstruktor festgelegten Reihenfolge. Außerdem gibt es den Nachrichtentyp `InvalidMessage`, der eine ungültige Nachricht definiert. Diese kann von dem Angreifer generiert und in eine Inbox geschrieben werden. Die ungültige Nachricht ist das abstrakte Äquivalent für den Fall, dass der Angreifer in der Realität ein übertragenes Byte-Array manipuliert und somit eine ungültige Byte-Array-Repräsentation entsteht. Dies ist wichtig für die Verfeinerungsbeziehung zwischen dem Code und formalem Modell (siehe Abschnitt 10.3.6). Die folgende Spezifikation zeigt einen Ausschnitt aus der `message`-Spezifikation der Kopierkartenanwendung.

```
data message =
mkLoad(. .amount : int;. .authterminal : HashedData) with isLoad |
mkRequestBalance with isRequestBalance |
..
mkInvalidMessage with isInvalidMessage;
```

Es wird für jede Nachrichtenklasse ein Prädikat generiert, mit dem getestet werden kann, ob ein Element von diesem Typ ist (z.B. das Prädikat `isLoad` für die Klasse `Load`).

Um Benutzernachrichten von anderen Nachrichten unterscheiden zu können, wird ein Prädikat `is_usermessage` generiert, das ein Element vom Typ `message` als Eingabe erhält. Dieses gibt `true` zurück, wenn die angegebene Nachricht eine Benutzernachricht ist. Das nachfolgende Listing zeigt die Axiomatisierung des Prädikats für die Kopierkartenanwendung.

```
is_usermessage(msg) ↔
    isURequestBalance(msg) ∨ isURequestCopies(msg)
  ∨ isUInsertMoney(msg) ∨ isUIssueCopies(msg)
  ∨ isUShowBalance(msg) ∨ isUResLoad(msg);
```

8. Stereotypen ≪HashData≫, ≪SignData≫ und ≪PlainData≫

Für die Stereotypen `HashData`, `SignData` und `PlainData` werden gleichnamige Datenspezifikationen erzeugt. Diese fassen alle Datentypen zusammen, die im UML-Modell mit den entsprechenden Stereotypen annotiert sind.

```
data HashData
  = wrapAuthData2HashData(. .authData : AuthData) with isAuthData;
```

Beispielhaft ist die Datenspezifikation `HashData` der Kopierkartenanwendung dargestellt. Für jede UML-Klasse, die mit dem Stereotyp ≪HashData≫ annotiert ist, wird ein Konstruktor bzw. Wrapper erzeugt. In dem konkreten Beispiel ist dies nur die Klasse `AuthData`. Die Datentypen `SignData` und `PlainData` werden analog generiert.

9. Listen

Abbildung 8.5 zeigt beispielhaft die Modellierung einer Liste mit UML (siehe Abschnitt 4.2.1.2).

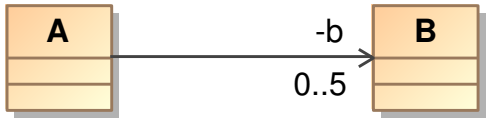

Abbildung 8.5.: Modellierung einer Liste in UML

Ein Assoziationsende mit Multiplizität größer als eins wird im formalen Modell in eine Liste übersetzt. Hierfür wird die in der Bibliothek des KIV-Systems vorhandene Listenspezifikation `list` aktualisiert. Der Typ der Listenelemente entspricht der UML-Datenklasse, auf die die Assoziation zeigt (`B`). Die Liste hat den Namen „`listOf` + Klassenname“, d.h. die im formalen Modell für das Beispiel generierte Liste heißt `listOfB`. Das Assoziationsende `b` hat somit im formalen Modell diesen Typ.

```
1  functions
2   add : listofB × B → listofB;
3   size : listOfB → int;
4   at : listOfB × int → B;
5   remove : listOfB × B → listOfB;
6
7  predicates
8   hasFree : listOfB;
9   contains : listOfB × B;
10
11 axioms
12  hasFree(listofb) ↔ #listofb < 5;
13  size(listofb) = #listofb;
14  contains(listofb, b) ↔ b ∈ listofb;
15  at(listofb, i) = listofb[i];
16  add(listofb, b) = listofb + b;
17  remove(listofb, b) = listofb -1l b;
```

Listing 8.2: Funktionen und Prädikate für den Listenzugriff

Die in MEL vordefinierten Listenoperationen werden in gleichnamige Funktionen und Prädikate übersetzt. Diese sind abhängig von dem Typ der in der Liste gespeicherten Elemente (`B`). Listing 8.2 zeigt die Funktionen und Prädikate für den Zugriff auf die Liste `listOfB`.

`listofb`, `b` und `i` sind Variablennamen. Die Länge einer Liste ist über das Prädikat `hasFree` festgelegt (Zeilen 8 und 12). Dieses liefert für Listen unbegrenzter Länge immer `true` zurück, für Listen begrenzter Länge nur dann, wenn die Liste noch nicht voll ist. Die Funktion `at` liefert das i-te Element der Liste (beginnend bei Index 0) (Zeilen 4 und 15). Ob der Index `i` gültig ist, d.h. sich an dieser Stelle in der Liste ein gültiges Element befindet, wird in der ASM vor Aufruf der Funktion (über den Test `i < size(listofb)`) geprüft. Die Funktion `add` fügt der Liste das Element `b` hinzu (Zeilen 2 und 16). Vor Aufruf der Funktion in der ASM wird geprüft, dass die Liste noch nicht voll ist (`hasFree(listofb)`). Die Funktion `remove` löscht das erste Vorkommen des Elements `b` aus der Liste (Zeilen 5 und 17). Dies geschieht mit Hilfe der in der KIV-Bibliothek vordefinierten Listenfunktion `-ll`.

10. Zustandsklasse einer Komponente

Eine UML-Klasse, die die möglichen Zustände einer Komponentenklasse beschreibt, wird bei der Generierung des formalen Modells in einen gleichnamigen abstrakten Datentyp übersetzt. Im folgenden Listing ist als Beispiel die Spezifikation der Zustandsklassen `StateCard` und `StateTerminal` der Kopierkartenanwendung angegeben. Diese sind als freie Datentypen mit konstanten Konstruktoren spezifiziert.

```
data StateCard = EXPLOAD | IDLE_CARD;
data StateTerminal =
   IDLE_TERMINAL | EXPRESPAY | EXPRESAUTH | EXPRESLOAD;
```

11. Klasse mit Stereotyp ≪Constants≫

Für eine UML-Klasse, die mit Stereotyp ≪Constants≫ annotiert ist, wird ebenfalls eine Datenspezifikation erzeugt. Diese definiert für jedes Attribut der UML-Klasse eine Konstante. Konstanten, für die im UML-Modell kein Typ angegeben ist, erhalten im formalen Modell den Typ `int`. Konstanten vom Typ `Number` (bzw. `int` im formalen Modell), für die im UML-Modell kein Wert angegeben ist, werden von eins an aufsteigend durchnummeriert. Konstanten vom Typ `Boolean`, für die kein Wert angegeben ist, erhalten den Wert `false`. Konstanten vom Typ `String` müssen bereits im UML-Modell einen Wert besitzen.

```
constants
 PAY : int;
 LOAD : int;

axioms
 PAY = 1;
 LOAD = 2;
```

Das Listing enthält beispielhaft die Datenspezifikation der Klasse `Constants` der Kopierkartenanwendung.

12. Manuelle Methoden

Der SecureMDD-Ansatz generiert für jede im UML-Modell definierte manuelle Operation eine Funktion (bzw. ein Prädikat für eine UML-Operation mit Rückgabetyp `Boolean`), die aber unspezifiziert bleibt. Die Verifikation von Sicherheitseigenschaften für eine Anwendung, die manuelle Methoden verwendet, ist nur mit Mehraufwand möglich. Deshalb sollte, wenn möglich, auf die Verwendung von manuellen Methoden verzichtet werden, wenn neben der Generierung des Quellcodes auch die Sicherheit der Anwendung verifiziert werden soll. Stattdessen sollte der Rumpf der Methode mitmodelliert werden (z.B. in einem Subdiagramm).

Enthält eine Anwendung eine manuelle Methode, muss derjenige, der die Verifikation durchführt, im formalen Modell von Hand entsprechende Axiome ergänzen, die die Funktion spezifizieren. Im generierten Code wird der Rumpf der manuellen Methode von Hand implementiert. Mit dem in KIV integrierten Java-Kalkül [180] muss dann verifiziert werden, dass die manuell erstellte Methodenimplementierung der im formalen Modell definierten Axiomatisierung genügt. Weiterhin muss sichergestellt werden, dass die manuelle Methodenimplementierung keine neuen Objekte erzeugt (da es für Java Card keine Garbage Collection gibt) und keine (im MEL vordefinierten) Methoden aufgerufen werden, die von dem Objektmanager verwaltete Objekte verwenden (da die Berechnung der Anzahl der verwalteten Objekte nur auf den in UML modellierten Protokollen basiert).

8.4.2. Transformation der Deploymentdiagramme

Die Transformation der Deploymentdiagramme lässt sich in zwei Teile untergliedern: Der Generierung der Kommunikationsinfrastruktur der Anwendung und der Spezifikation des Angreifers. Beide Aspekte werden im Folgenden erläutert.

8.4.2.1. Generierung der Kommunikationsinfrastruktur

In Abschnitt 8.2 wurde bereits erläutert, dass ein UML-Kommunikationspfad (im Deploymentdiagramm) im formalen Modell durch eine Verbindung (`Connection`) repräsentiert wird. Eine Verbindung besteht aus zwei Endpunkten. Ein Endpunkt ist ein Tupel aus einem Agenten und einem `Port`. Nachfolgend sind die Datenspezifikationen für Ports, Endpunkte und Verbindungen sowie wichtige Prädikate angegeben.

Ports:
Für die Ports der modellierten Anwendung wird eine Datenspezifikation mit Namen `ports` generiert. Diese ist eine Aufzählung über alle im Deploymentdiagramm vorhandenen Ports der Anwendung. Dabei werden sowohl die im Deploymentdiagramm benannten als auch die unbenannten Ports berücksichtigt. Für die im UML-Modell unbenannten Ports werden eindeutige Namen vergeben. Diese ergeben sich aus den Namen der Knoten (UML-`Nodes`) der zugehörigen Verbindung. Für die benannten Ports werden die im Deploymentdiagramm angegebenen Namen übernommen.

```
data ports
= Copycard2CopyingMachineDefaultPort |
  CopyingMachine2CopycardDefaultPort |
  CopyingMachine2CardOwnerDefaultPort |
  CardOwner2CopyingMachineDefaultPort |
  Copycard2DepositMachineDefaultPort |
  DepositMachine2CopycardDefaultPort |
  DepositMachine2CardOwnerDefaultPort |
  CardOwner2DepositMachineDefaultPort;
```

Das Listing zeigt die Datenspezifikation `ports` für die Kopierkartenanwendung, in der im Deploymentdiagramm alle Ports unbenannt sind (siehe Abschnitt 4.2.2.1).

Abbildung 8.6 illustriert die Zuordnung der für die Ports der Kopierkartenanwendung automatisch generierten Namen zu den entsprechenden Endpunkten.

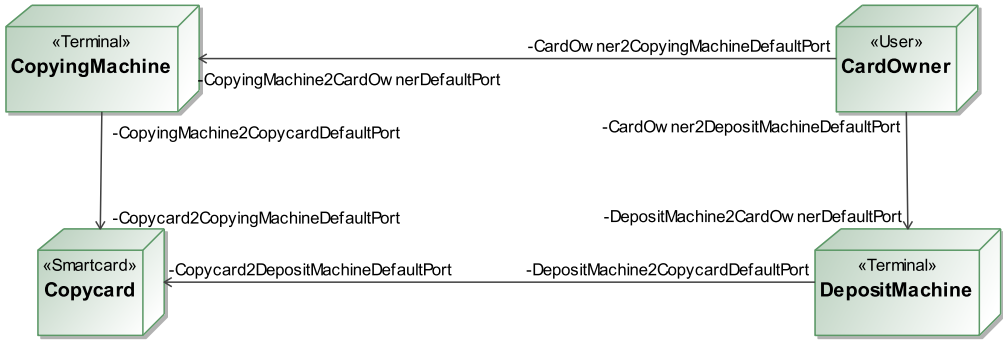

Abbildung 8.6.: Automatische Benennung der Ports der Kopierkartenanwendung im formalen Modell

Endpunkt:
Ein Endpunkt besteht aus einem Agenten sowie einem Port. Die Spezifikation ist unabhängig von der konkreten Anwendung und nachfolgend abgebildet.

```
data endpoint =  . ⊙ . ( . .agent : agent; . .port : ports);
```

In dem Prädikat `endpoint-ok` ist festgelegt, welche gültigen Endpunkte es für die modellierte Anwendung gibt. Ein Endpoint ist gültig, wenn der Agententyp sowie der Port zusammenpassen, z.B. muss in der Kopierkartenanwendung der Agent des Ports `CardOwner2DepositMachineDefaultPort` vom Typ `User` sein. Dies sind genau die Endpunkte, die während eines im UML-Modell definierten Protokolllaufs miteinander kommunizieren, d.h. sie definieren das intendierte Kommunikationsverhalten einer Anwendung. Zusätzlich gibt es jedoch noch einige weitere gültige Verbindungen und somit auch weitere gültige Endpunkte. Denn im praktischen Einsatz ist es möglich, dass bei einer Anwendung mit verschiedenen Smart Card-Komponenten eine Karte in ein „falsches“ Lesegerät gesteckt wird. Zum Beispiel kann es bei der praktischen Verwendung der „elektronischen Gesundheitskarte“

passieren, dass eine Gesundheitskarte in das Lesegrät für den Heilberufsausweis des Arztes gesteckt wird und auf diese Weise mit einem `ArztPC` kommuniziert. Dieses Verhalten ist nicht intendiert und somit nicht explizit in den Deployment- und Aktivitätsdiagrammen modelliert. Da es in der Praxis jedoch vorkommen kann (und eventuell zusätzliche Angriffsmöglichkeiten bietet), muss das Prädikat auch diese (eigentlich nicht gewünschten) Endpunkte zulassen. Die (anwendungsspezifische) Axiomatisierung des Prädikats `endpoint-ok` ist in Listing 8.3 (für das Deploymentdiagramm in Abbildung 8.7) abgebildet.

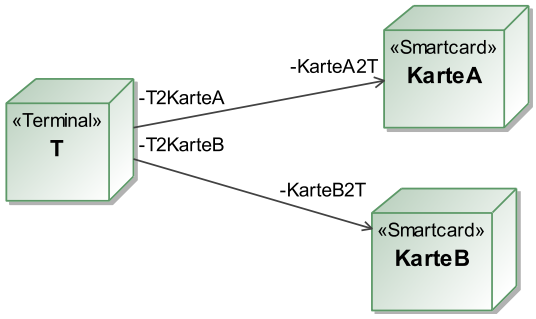

Abbildung 8.7.: Beispiel für ein Deploymentdiagramm mit zwei Kartentypen

In dem Beispiel gibt es zwei verschiedene Kartentypen (`KarteA` und `KarteB`), die mit einem Terminal vom Typ `T` kommunizieren. Für ein besseres Verständnis sind für alle Ports des Diagramms Namen angegeben.

```
1  predicates
2   endpoint-ok : endpoint;
3
4  variables
5   endp : endpoint;
6
7  axioms
8   endpoint-ok(endp) ↔
9      is_T(endp .agent)
10     ∧ endp .port = T2KarteA
11   ∨ is_T(endp .agent)
12     ∧ endp .port = T2KarteB
13   ∨ is_KarteA(endp .agent)
14     ∧ endp .port = KarteA2T
15   ∨ is_KarteB(endp .agent)
16     ∧ endp .port = KarteB2T
17   ∨ is_KarteB(endp .agent)
18     ∧ endp .port = KarteA2T
19   ∨ is_KarteA(endp .agent)
20     ∧ endp .port = KarteB2T
```

Listing 8.3: Axiomatisierung des Prädikats `endpoint-ok`

Das Prädikat `endpoint-ok` definiert jeden im UML-Diagramm dargestellten Endpunkt als gültig (Zeilen 5-13). Zusätzlich sind die Endpunkte, die ausdrücken, dass eine Karte in einem „falschen“ Lesegerät steckt, gültige Endpunkte. In dem angegebenen Beispiel ist dies der Fall, wenn sich eine `KarteB` im Lesegerät für `KarteA` befindet (Zeilen 14-15) oder sich eine `KarteA` im Lesegerät für `KarteB` befindet (Zeilen 16-17).

Verbindung:
Eine Verbindung besteht aus zwei Endpunkten. Die Spezifikation ist unabhängig von der konkreten Anwendung und nachfolgend abgebildet.

```
data connection
 = mk-connection( . .endpoint1 : endpoint;
                  . .endpoint2 : endpoint);
```

Das Prädikat `conn-ok` definiert die gültigen Verbindungen der modellierten Anwendung. Eine Verbindung ist gültig, wenn ihre Endpunkte gültig sind (d.h. `endpoint-ok` für beide Endpunkte gilt) und die Ports der zwei Endpunkte im UML-Deploymentdiagramm zu demselben UML-Kommunikationspfad gehören. Somit ist das Prädikat `conn-ok` auch für einen Endpunkt wahr, wenn eine Smart Card in einem „falschen“ Lesegerät steckt.

In Listing 8.4 sind das Prädikat `conn-ok` sowie die zugehörige Axiomatisierung (am Beispiel der Kopierkartenanwendung) angegeben.

```
predicates
conn-ok : connection;

axioms
conn-ok(conn) ↔
    endpoint-ok(conn .endpoint1)
 ∧  endpoint-ok(conn .endpoint2)
 ∧          (conn .endpoint1 .port
            = CardOwner2CopyingMachineDefaultPort
         ∧  conn .endpoint2 .port
            = CopyingMachine2CardOwnerDefaultPort
     ∨      conn .endpoint1 .port
            = CardOwner2DepositMachineDefaultPort
         ∧  conn .endpoint2 .port
            = DepositMachine2CardOwnerDefaultPort
     ∨      ...)
```

Listing 8.4: Das Prädikat `conn-ok`

Aktuell aufgebaute Verbindungen:
Das Kommunikationsmodell der formalen Spezifikation sieht vor, dass eine Nachricht nur dann über eine Verbindung verschickt werden kann, wenn diese Verbindung zuvor explizit aufgebaut (und nicht wieder beendet) wurde. Ein Endpunkt kann zu einem Zeitpunkt nur an einer aufgebauten Verbindung beteiligt sein. Das explizite Auf- und Abbauen von Verbindungen verhindert, dass eine Smart Card zu einem Zeitpunkt mit mehreren Terminals kommunizieren

kann. Dies ist in der Realität ebenfalls nicht möglich, da sich eine Smart Card zu einem Zeitpunkt nur in maximal einem Lesegerät (das an ein Terminal angeschlossen ist) befinden kann. Außerdem ist es in der Realität nicht möglich, dass sich zu einem Zeitpunkt in einem Kartenleser mehrere Karten befinden. Dies wird im formalen Modell durch den expliziten Verbindungsauf- und abbau ebenfalls ausgeschlossen.

Die aktuell aufgebauten Verbindungen werden in der Menge `connections` gespeichert. Dies ist eine Aktualisierung der Mengenspezifikation `set`, die in der KIV-Basisbibliothek definiert ist.

Das Aufbauen einer neuen bzw. das Abbauen einer bestehenden Verbindung ist über zwei ASM-Regeln CONNECT und DISCONNECT realisiert, die Teil der ASM sind (siehe Abschnitt 8.4.3). Das Prädikat `connect-possible(conn, connections)` prüft, ob der Aufbau der Verbindung `conn` aktuell möglich ist.

```
connect-possible(conn, connections) ↔
   ¬ connected(conn .endpoint1, connections)
 ∧ ¬ connected(conn .endpoint2, connections)
 ∧ valid-conn(conn)
```

Der Aufbau ist möglich, wenn beide Endpunkte nicht an einer bestehenden Verbindung beteiligt sind (d.h. das Prädikat `connected` für beide Endpunkte `false` zurückgibt) und die Verbindung `conn` gültig ist (d.h. das Prädikat `valid-conn` wahr ist). Beide Prädikate sind im Folgenden erläutert.

```
connected(endp1, connections) ↔
(∃ endp2. mk-connection(endp1, endp2) ∈ connections
   ∨ mk-connection(endp2, endp1) ∈ connections);
```

Das Prädikat `connected` gibt `true` zurück, wenn der Endpunkt `endp1` bereits an einer aktuell aufgebauten Verbindung (die in `connections` gespeichert ist) beteiligt ist.

```
predicates
valid-conn : connection;

axioms
valid-conn(conn) ↔
   conn-ok(conn)
 ∧ exagent(conn .endpoint1 .agent)
 ∧ exagent(conn .endpoint2 .agent);
```

Das Prädikat `valid-conn` gibt `true` zurück, wenn `conn-ok` gilt und die an der Verbindung beteiligten Agenten gültig sind (d.h. das Prädikat `exagent` für beide beteiligten Agenten `true` zurückgibt). Das Prädikat `exagent` prüft, ob der Name `name` vom Typ `nat` kleiner oder gleich der Konstante ist, die die Anzahl der Agenten dieses Typs beschreibt.

Inbox:
Jeder Endpunkt besitzt eine Inbox, die die an einen Agenten gesendeten, aber noch nicht abge-

rufenen Nachrichten speichert. Eine Inbox ist eine Funktion mit der Signatur `agent` → `ports` → `messagelist`. Eine `messagelist` ist eine Aktualisierung der in der KIV-Bibliothek definierten Listen mit Element `message`. Die Auswertung der Funktion `inputs(agent)(port)` gibt die Liste der für diesen Agenten und Port gespeicherten Nachrichten zurück. Die Funktion `add` fügt eine neue Nachricht an das Ende einer Inbox an, die Funktion `remove` löscht die erste Nachricht aus einer Inbox. Das Prädikat `msgininputs` gibt für eine Nachricht `msg` an, ob sich diese aktuell in einer Inbox befindet.

Mit Ausnahme der Spezifikation `ports` und der Prädikate `endpoint-ok` und `conn-ok` ist die Formalisierung der Kommunikationsstruktur unabhängig von der konkreten Anwendung. Im Folgenden werden die Übersetzung der im Deploymentdiagramm angegebenen Angreiferfähigkeiten sowie die formale Spezifikation des Angreifers erläutert.

8.4.2.2. Generierung der Spezifikation des Angreifers

Der Stereotyp ≪Threat≫ bzw. dessen Tags `read`, `send` und `suppress` werden in drei gleichnamige Prädikate übersetzt. Diese geben für eine Verbindung an, ob der Angreifer diese abhören kann bzw. Nachrichten verschicken oder unterdrücken kann. Die Prädikate sind abhängig von der konkreten Anwendung und geben `true` zurück, wenn der zugehörige Kommunikationspfad im UML-Deploymentdiagramm entsprechend annotiert sind. Listing 8.5 gibt beispielhaft die Axiomatisierung des Prädikats `send` für die Kopierkartenanwendung an. Die beiden anderen Prädikate sind analog spezifiziert.

```
predicates
send : connection;

axioms
send(conn) ↔
        conn .endpoint1 .port = CopyingMachine2CopycardDefaultPort
    ∧   conn .endpoint2 .port = Copycard2CopyingMachineDefaultPort
∨       conn .endpoint1 .port = DepositMachine2CopycardDefaultPort
    ∧   conn .endpoint2 .port = Copycard2DepositMachineDefaultPort
```

Listing 8.5: Axiomatisierung des Prädikats `send`

Zusätzlich geben die Prädikate `can-read`, `can-send` und `can-suppress` an, ob der Angreifer aktuell eine Nachricht, die über eine Verbindung `conn` gesendet wird, abhören kann bzw. über diese Verbindung eine Nachricht senden bzw. unterdrücken kann. Dies ist möglich, wenn die gewählte Verbindung `conn` zu diesem Zeitpunkt existiert (d.h. `conn` ∈ `connections`) und das Prädikat `read`, `send` bzw. `suppress` `true` zurückgibt.

Der Angreifer ist ebenfalls als Agententyp in der Spezifikation `agent` definiert, es gibt jedoch nur eine Instanz dieses Typs. Das Wissen des Angreifers ist explizit in einer Menge vom Typ `attackerdataset` gespeichert, das Elemente vom Typ `attackerdata` enthält. Diese Menge ist eine Aktualisierung der in der KIV-Bibliothek definierten Menge `set`. Der frei erzeugte Datentyp `attackerdata` ist abhängig von der entwickelten Anwendung. Ein Element vom Typ `attackerdata` kann eine Nachricht, ein Element vom Typ `data` (d.h.

die Übersetzung einer UML-Datenklasse), ein Element vom Typ `HashData`, `SignData` oder `PlainData` sowie ein vordefinierter Sicherheitsdatentyp sein.

Im Folgenden ist der Typ `attackerdata` der Kopierkartenanwendung abgebildet.

```
data attackerdata =
  amessage(. .message : message) with is_amessage
| adata(. .d : Data) with is_adata
| ahashdata(. .hashdata : HashData) with is_ahashdata
| ahasheddata(. .hasheddata : HashedData) with is_ahasheddata
| asecret(. .secret : Secret) with is_asecret
| anonce(. .nonce : Nonce) with is_anonce
```

Auch an dieser Stelle zeigt sich, dass das formale Modell auf die konkrete Anwendung zugeschnitten ist. Der Datentyp `attackerdata` der Kopierkartenanwendung kann Nachrichten (vom Typ `message`), Elemente vom Typ `data` sowie `hashdata`, Hashwerte (vom Typ `HashedData`) sowie Secrets und Nonces speichern. Dies sind genau die Datentypen und UML-Klassen, die in der Kopierkartenanwendung vorkommen. Da diese Anwendung keine Verschlüsselung sowie keine digitalen Signaturen verwendet, müssen die Datentypen `EncDataSymm`, `EncDataAsymm` und `SignedData` im Angreiferwissen auch nicht berücksichtigt werden. Dies macht die Spezifikation sowie den Umgang mit dem Angreiferwissen bei der Verifikation erheblich einfacher.

Wird eine Nachricht über eine Verbindung gesendet, die der Angreifer abhören kann, wird diese vor dem Schreiben in die Inbox des Empfängers in das Angreiferwissen aufgenommen. Dies ist über eine Funktion $\int+$ realisiert, die das aktuelle Angreiferwissen sowie die mitgelesene Nachricht(enliste) als Eingabe bekommt, daraus neues Wissen ableitet und dieses dem Angreiferwissen hinzufügt. Enthält eine Nachricht zum Beispiel ein verschlüsseltes Datum vom Typ `EncDataSymm`, wird dieses dem Angreiferwissen hinzugefügt. Damit ist die Berechnung jedoch noch nicht beendet. Ist der Schlüssel, mit dem das verschlüsselte Datum entschlüsselt werden kann, bereits im Angreiferwissen enthalten, kann der Angreifer dieses Datum entschlüsseln und so weiteres Wissen gewinnen. Es muss deshalb rekursiv berechnet werden, ob der Angreifer durch Hinzufügen des entschlüsselten Datums weitere Daten lernen kann. Die Berechnung endet, wenn alle möglichen Daten aus dem aktuellen Angreiferwissen extrahiert wurden. Dies bedeutet, dass ein Fixpunkt über der Menge vom Typ `attackerdataset` bezüglich der (anwendungsabhängigen) Analyseregeln erreicht wurde. Dieser Fixpunkt existiert immer und kann nach endlich vielen Schritten erreicht werden. Die Funktion $\int+$ ist somit wohldefiniert. Folgendes Listing zeigt die Spezifikation der Funktion $\int+$.

```
1 functions
2 . ∫+ . : attackerdataset × message → attackerdataset;
3 . ∫+ . : attackerdataset × messagelist → attackerdataset;
4
5 axioms
6 adset ∫+ msg = adset ⋃ analyze(msg, adset);
7 adset ∫+ [] = adset;
8 adset ∫+ (msg + msgs) = (adset ⋃ analyze(msg, adset)) ∫+ msgs;
```

Eine Nachricht wird dem Angreiferwissen hinzugefügt, indem die Vereinigung über dem alten, bisherigen Angreiferwissen und dem Ergebnis der `analyze`-Funktion berechnet wird (Zeile 6). Wird eine Liste von Nachrichten dem Wissen hinzugefügt, wird die Vereinigung rekursiv über alle Nachrichten berechnet (Zeile 7-8).

Listing 8.6 zeigt beispielhaft einen Teil der Spezifikation der Funktion `analyze`. Diese berechnet das Wissen, das sich aus einer Nachricht bzw. einem Element vom Typ `AuthData` und dem aktuellen Angreiferwissen ableiten lässt.

```
1  functions
2  analyze : attackerdata × attackerdataset → attackerdataset;
3
4  axioms
5  analyze(amessage(mkLoad(i, hasheddata)),adset) =
6  {ahasheddata(hasheddata)};
7  analyze((adata(wrapAuthData2data(authdata)), adset) =
8  {asecret(authdata.passphrase)} ⋃ {anonce(authdata.challenge)};
9  analyze(amessage(mkInvalidMessage), adset) = ∅;
10 ...
```

Listing 8.6: Ausschnitt aus der Spezifikation der Funktion `analyze`

Die Funktion `analyze` hat zwei Eingabeparameter, einen vom Typ `attackerdata` und einen vom Typ `attackerdataset` (Zeile 2). Wird die Funktion beispielsweise auf eine Nachricht vom Typ `Load` angewendet, wird der in der Nachricht enthaltene Hashwert im Angreiferwissen gespeichert (Zeile 5-6). Der Hashwert ist jedoch nicht weiter zerlegbar, d.h. der Angreifer lernt nicht den Klartext, aus dem der Hashwert gebildet wurde. Der in der `Load`-Nachricht enthaltene Integer wird nicht in das Angreiferwissen aufgenommen, da der Angreifer zu jeder Zeit einen beliebigen Integer erzeugen kann. Wird die `analyze`-Funktion auf einem Element vom Typ `AuthData` aufgerufen, werden das in dem Element enthaltenen Secret `passphrase` sowie die Nonce `challenge` dem Angreiferwissen hinzugefügt (Zeilen 7-8). Die weiteren Felder des Datentyps (vom Typ `int`) werden bei der Berechnung des Angreiferwissens ignoriert.

Wenn die modellierte Anwendung Verschlüsselung verwendet, ist die Berechnung des Angreiferwissens komplizierter. In diesem Fall ist es möglich, dass eine Nachricht einen Schlüssel enthält. Dieser ermöglicht z.B. die Entschlüsselung eines verschlüsselten Datums, das sich bereits im Angreiferwissen befindet. Durch die Entschlüsselung könnten dem Angreifer weitere sicherheitsrelevante Daten wie Schlüssel oder Geheimnisse bekannt werden, die in dem entschlüsselten Dokument enthalten sind. In diesem Fall wird eine `analyze`-Funktion benötigt, die dem Angreiferwissen sämtliche Elemente hinzufügt, die der Angreifer durch Bekanntwerden des Schlüssels lernt.

Möchte der Angreifer eine Nachricht über einen UML-Kommunikationspfad (der mit dem Tag `send` annotiert ist) senden, muss er diese Nachricht mit Elementen aus seinem Wissen erstellen. Um zu spezifizieren, ob ein Angreifer ein Datum erzeugen kann, wird ein Prädikat $\gg$ generiert. Listing 8.7 zeigt einen Teil der anwendungsabhängigen Spezifikation.

Das Prädikat hat zwei Eingabeparameter, das aktuelle Angreiferwissen sowie ein Element vom Typ `attackerdata`, das der Angreifer erzeugen möchte (Zeile 2). Das Prädikat sagt

```
1  predicates
2  . ≫ . : attackerdataset × attackerdata;
3
4  axioms
5  adset ≫ amessage(mkLoad(i,hasheddata)) ↔
6                          adset ≫ ahasheddata(hasheddata);
7  adset ≫ ahasheddata(hasheddata) ↔
8                          ahasheddata(hasheddata) ∈ adset
9                        ∨ adset ≫ ahashdata(hasheddata.hash);
10 adset ≫ amessage(mkInvalidMessage);
11 ...
```

Listing 8.7: Ausschnitt aus der Spezifikation `Generate` der Kopierkartenanwendung

aus, ob dieses Element aus dem aktuellen Wissen erzeugbar ist. Eine `Load`-Nachricht zum Beispiel, die den Integer `i` und den Hashwert `hasheddata` enthält, ist aus dem Angreiferwissen ableitbar, wenn der Angreifer den Hashwert `hasheddata` aus seinem aktuellen Wissen ableiten kann (Zeile 5-6). Dies ist der Fall, wenn der Hashwert oder der zugehörige Klartext (der im formalen Modell im Datentyp `HashedData` gespeichert ist) im Angreiferwissen enthalten sind (Zeile 7-9). Nichtkryptographische Daten wie Integer, Strings oder boolesche Werte sowie eine Nachricht vom Typ `InvalidMessage` kann der Angreifer immer erzeugen, unabhängig von seinem aktuellen Wissen.

Da in der Kopierkartenanwendung keine Verschlüsselung und Signaturen verwendet werden, müssen diese auch bei der Berechnung des Angreiferwissens bzw. der ableitbaren Nachrichten nicht berücksichtigt werden. Das Resultat ist ein kleineres und übersichtlicheres formales Modell und auch die Verifikation wird aufgrund dieser Optimierung deutlich einfacher.

Eine detaillierte Erläuterung der Funktionen $\int+$ und `analyze` sowie des Prädikats $\gg$ ist in [78] (Abschnitte 7.2 und 7.3) zu finden. Eine ähnliche Definition wurde von Paulson in [152] gemacht.

8.4.3. Transformation der Aktivitätsdiagramme

In diesem Abschnitt wird die Übersetzung des dynamischen Verhaltens erläutert. Dieses ist im UML-Modell durch UML-Aktivitätsdiagramme beschrieben, die durch Ausdrücke der Sprache MEL erweitert werden. Im formalen Modell ist das dynamische Verhalten durch eine Abstract State Machine (ASM) [25, 77] definiert. Im Folgenden wird zunächst der allgemeine Aufbau der ASM beschrieben. Anschließend wird die Übersetzung der Aktivitätsdiagramme, d.h. der in den Diagrammen verwendeten UML-Elemente und die Übersetzung der MEL-Ausdrücke, erläutert.

8.4.3.1. Allgemeiner Aufbau der Abstract State Machine

Die ASM ist in KIV als eine Prozedur angegeben, die aus einer `while`-Schleife besteht. Die Prozedur besitzt Parameter für alle dynamischen Funktionen, die aus den Attributen und Assozaitionsenden der Komponentenklassen generiert wurden (siehe Abschnitt 8.4.1). Außer-

dem hat sie Parameter für die Variablen `all-nonces` und `next-nonce` (für das Generieren einer neuen Nonce bzw. eines neuen Schlüssels), `inputs` (für die Inboxen), `connections` (für die bestehenden Verbindungen), `attacker-known` (das Angreiferwissen) und `stop` (eine boolesche Variable). Die Spezifikation der ASM ist nachfolgend abgebildet.

```
procedures
ASM   : ...

declaration
ASM(...) {
   while (¬ stop) {
      STEP(..);
      stop := [?];
   }
}
```

Die ASM ruft in einer `while`-Schleife die ASM-Regel STEP auf. Nachdem dieser Aufruf beendet wurde, wird nichtdeterministisch gewählt, ob die ASM einen weiteren Schritt ausführen soll. Wird für eine Variable `stop` der Wert `false` gewählt, wird die Schleife erneut durchlaufen, ansonsten wird die Ausführung der ASM beendet.

Für jede im Klassendiagramm definierte Komponentenklasse wird ein eigener `asm-step` erzeugt. Abstrakte Oberklassen einer Komponente werden bei der Generierung der formalen Spezifikation, ebenso wie bei der Codegenerierung, eliminiert. Dies bedeutet, dass ein `asm-step` für jede konkrete Subklasse, aber nicht für die Oberklasse definiert wird. Zusätzlich gibt es für jede Anwendung die `asm-steps` `connect` (zum Aufbau einer neuen Verbindung), `disconnect` (zum Abbau einer bestehenden Verbindung), `user-agent-step` (für einen Schritt eines Benutzers) sowie `attacker-agent-step` (für einen Schritt des Angreifers). Die Spezifikation `asm-step` der Kopierkartenanwendung ist nachfolgend angegeben.

```
data asm-step
= connect | disconnect | attacker-agent-step |
  user-agent-step | Copycard-agent-step |
  DepositMachine-agent-step | CopyingMachine-agent-step;
```

Die ASM-Regel STEP wählt nichtdeterministisch aus, welcher Agent (oder die Kommunikationsstruktur) als nächstes eine Schritt durchführen soll. Möglich sind folgende Schritte:

- der Schritt CONNECT wählt zufällig eine Verbindung aus, die zum aktuellen Zeitpunkt aufgebaut werden kann und fügt sie der Menge der `connections` hinzu.
- der Schritt DISCONNECT wählt zufällig eine bestehende Verbindung aus der Menge `connections` aus und löscht sie aus der Menge. Das heißt, in dem Schritt wird eine bestehende Verbindung unterbrochen. Dies berücksichtigt zum Beispiel den im praktischen Einsatz der Anwendung möglichen Fall, dass eine Smart Card plötzlich aus einem Kartenleser gezogen wird und somit keine Verbindung mehr zwischen der Karte und dem Terminal besteht.

- der Schritt ATTACKER führt einen Schritt des Angreifers durch, in dem er zufällig wählt, ob der Angreifer in diesem Schritt eine (aus seinem Wissen abgeleitete) Nachricht senden, eine Nachricht unterdrücken oder eine bestehende Verbindung beenden soll. Die gewählte Aktion wird anschließend durchgeführt.
- der Schritt USER wählt nichtdeterministisch einen Agenten vom Typ `user` aus und führt für diesen einen Schritt aus. Ein USER-Schritt ist entweder das Lesen einer Nachricht aus einer (nichtleeren) Inbox (d.h. der Empfang einer Nachricht, die an diesen Benutzer geschickt wurde) oder das Versenden einer Nachricht an ein Terminal (und somit der Start eines neuen Protokolllaufs).
- für jede im UML-Modell definierte Smart Card-Komponente wird ebenfalls ein STEP erzeugt. Dieser wählt nichtdeterministisch einen Agenten dieses Typs aus, entnimmt eine Nachricht aus einer (zufällig gewählten, nichtleeren) Inbox des Agenten und verarbeitet die Nachricht, d.h. führt einen Protokollschritt aus.
- für jede im UML-Klassendiagramm definierte Terminalkomponente wird ebenfalls ein STEP erzeugt. Dieser ist analog zu einem Smart Card-STEP spezifiziert.

Die Spezifikation STEP ist in Listing 8.8 beispielhaft für die Kopierkartenanwendung dargestellt.

```
1  STEP(..) : {
2  let asm-step = [?], exception_occurred = false
3   in if asm-step = connect
4      then CONNECT(..)
5      else if asm-step = disconnect
6           then DISCONNECT(..)
7           else if asm-step = attacker-agent-step
8                then ATTACKER(..)
9                else if asm-step = user-agent-step
10                    then choose  ag
11                         with (is_user(ag) ∧ exagent(ag))
12                         in USER(..)
13                         ifnone skip
14                    else if asm-step = Copycard-agent-step
15                         then choose  ag
16                              with (is_Copycard(ag)
17                                    ∧ exagent(ag))
18                              in COPYCARDSTEP
19                              ifnone skip
20                         else if asm-step =
21                                 DepositMachine-agent-step
22                              then choose ag
23                                   with (is_DepositMachine(ag)
24                                         ∧ exagent(ag))
25                                   in DEPOSITMACHINESTEP
26                                   ifnone skip
27                              else if asm-step =
28                                     CopyingMachine-agent-step
29                                   then choose ag
30                                        with (
31                                           is_CopyingMachine(ag)
32                                           ∧ exagent(ag))
33                                        in COPYINGMACHINESTEP
34                                        ifnone skip;};
```

Listing 8.8: Die Spezifikation STEP der Kopierkartenanwendung

Ist der nichtdeterministisch gewählte `asm-step` ein `user-step`, wird zunächst nichtdeterministisch ein Agent ausgewählt, für den ein Schritt durchgeführt werden soll (Zeilen 9-13). Dies gilt auch für alle anderen Agententypen, von denen es mehr als eine Instanz gibt. Das Prädikat `exagent` stellt dabei sicher, dass es diese Instanz auch wirklich gibt, d.h. dass sein Name (vom Typ `nat`) kleiner ist als die Anzahl der Agenten dieses Typs, die über eine Konstante festgelegt ist. Alle Agenten des Typs User, deren Name größer oder gleich null und kleiner gleich der Konstanten ist, sind gültige Instanzen dieses Typs. Anschließend wird die ASM-Regel USER aufgerufen, die den Benutzerschritt für den gewählten Agenten durchführt. Existiert kein gültiger Agent vom Typ User, wird in diesem Schritt nichts gemacht (`ifnone`

skip). Für jede Smart Card-Komponente (in diesem Fall nur die Copycard) und jede Terminalkomponente (d.h. die Komponenten DepositMachine und CopyingMachine) werden analog zu dem Agententyp User entsprechende agent-steps generiert.

Im Folgenden werden die Regeln CONNECT, DISCONNECT, ATTACKER und USER sowie die Regeln für die Smart Card- und Terminalkomponenten erläutert. Sie sind im KIV-System ebenfalls jeweils als Prozedur spezifiziert.

Regel CONNECT:
Die Regel CONNECT ist unabhängig von der konkreten Anwendung und hat die Ein- und Ausgabeparameter connections (vom Typ connections) und inputs (vom Typ Agent → ports → messagelist).

```
1  procedures
2  CONNECT : connections × (agent → ports → messagelist);
3
4  declaration
5  CONNECT(inputs;connections) {
6  choose  conn
7  with connect-possible(conn, connections, inputs)
8  in connections := connections ++ conn
9  ifnone skip
10 };
```

Die Regel wählt zunächst nichtdeterministisch eine Verbindung conn aus, für die der Aufbau einer Verbindung möglich ist (Zeilen 6-7). Diese Verbindung wird anschließend zur Menge der bestehenden Verbindungen (connections) hinzugefügt (Zeile 8). Kann keine Verbindung gefunden werden, die aktuell aufgebaut werden kann, ist der ASM-Schritt beendet (Zeile 9).

Regel DISCONNECT:
Die ASM-Regel DISCONNECT ist ebenfalls unabhängig von der konkreten Anwendung und hat dieselben Ein-/Ausgabeparameter wie die Regel CONNECT.

```
1  procedures
2  DISCONNECT : connections × (agent → ports → messagelist);
3
4  declaration
5  DISCONNECT(;connections, inputs){
6  choose conn
7  with disconnect-possible(conn, connections, inputs)
8  in {connections := connections -- conn ;
9      inputs(conn .endpoint1 .agent)(conn .endpoint1 .port):=[];
10     inputs(conn .endpoint2 .agent)(conn .endpoint2 .port):=[]}
11 ifnone skip };
```

Die Regel wählt zunächst nichtdeterministisch eine Verbindung aus, die getrennt werden kann (Zeilen 6-7). Dies ist der Fall, wenn das Prädikat disconnect-possible true zurückgibt (d.h. die Verbindung ist aktuell aufgebaut). Die gewählte Verbindung wird dann aus der

Menge connections gelöscht und die Inboxen der beiden Endpunkte werden geleert (Zeilen 8-10). Kann keine Verbindung conn gefunden werden, die abgebaut werden kann, ist der ASM-Schritt beendet (Zeile 11).

Regel ATTACKER:
Die Regel ATTACKER hat die Ein-/Ausgabeparameter connections (vom Typ connections, die bestehenden Verbindungen), attacker-known (vom Typ attackerdataset, das Angreiferwissen) und inputs (vom Typ agent → ports → messagelist, die Inboxen der Agenten).

```
1 ATTACKER(connections, attacker-known; inputs) {
2 let attacker-step = [?]
3 in if attacker-step = attacker-send
4    then ATTACKER-SEND(connections, attacker-known; inputs)
5    else if attacker-step = attacker-suppress
6         then ATTACKER-SUPPRESS(connections; inputs)};
```

Die Regel wählt zunächst indeterministisch aus, welche Art von Angreiferschritt durchgeführt werden soll (Zeile 2) und führt den gewählten Schritt dann aus (Zeile 3-6). Mögliche Schritte sind das Senden einer Nachricht oder das Unterdrücken einer Nachricht. Beide Schritte sind im KIV-System als Prozedur spezifiziert. ATTACKER-SEND wählt zunächst nichtdeterministisch einen Endpunkt aus, über deren zugehörige bestehende Verbindung er Nachrichten verschicken darf. Außerdem wird nichtdeterministisch eine Nachricht gewählt, die der Angreifer aus seinem Wissen ableiten kann und diese dann an das Ende der Inbox des gewählten Ports geschrieben. ATTACKER-SUPPRESS wählt nichtdeterministisch einen Endpunkt aus, für deren zugehörige Verbindung er Nachrichten unterdrücken kann und dessen Inbox nicht leer ist. Anschließend wird die erste Nachricht in dieser Inbox gelöscht.

Regel USER:
Die Regel USER hat die Eingabeparameter agent (vom Typ agent, der zuvor gewählte Agent vom Typ user) und connections (vom Typ connections) sowie die Ein-/Ausgabeparameter attackerdataset (das Angreiferwissen) und inputs (die Inboxen).

```
procedures
USER : agent × connections; attackerdataset ×
            (agent → ports → messagelist);

declaration
USER (ag, connections; attacker-known, inputs){
let user-step = [?]
in if user-step = send
   then USER-SEND(ag, connections; attacker-known, inputs)
   else if user-step = read
        then USER-READ(ag, connections; inputs)};
```

Die Regel wählt nichtdeterministisch aus, welche Art von Benutzerschritt durchgeführt werden soll. Möglich sind das Senden einer Nachricht oder das Lesen einer Nachricht, die sich in

einer Inbox des gewählten Agenten befindet. Beide sind im KIV-System als Prozedur spezifiziert. USER-SEND wählt nichtdeterministisch einen zu dem gewählten Agenten gehörenden Port aus, für den aktuell eine Verbindung besteht. Außerdem wird nichtdeterministisch eine Benutzernachricht (für die das Prädikat `is_usermessage` gilt) gewählt. Diese wird dann anschließend durch Aufruf der Prozedur SEND (siehe Seite 225) versendet. Durch das Versenden einer neuen Benutzernachricht wird ein neuer Protokolllauf begonnen. USER-READ wählt zunächst ebenfalls nichtdeterministisch einen Port des gewählten Agenten aus, für den es eine existierende Verbindung gibt. Anschließend wird die zu dem Port gehörende Inbox geleert. Dies ist die abstrakte Modellierung dafür, dass der Benutzer eine Benutzernachricht, die das Terminal an ihn geschickt hat, gelesen hat. Dies spiegelt wider, dass der Benutzer im SecureMDD-Ansatz lediglich Nachrichten an einem Terminal eingeben oder von einem Terminal empfangen kann. Er selber, als Person, kann hingegen keine Dokumente verschlüsseln oder Schlüssel erzeugen.

Komponentenschritt:
Für jede Smart Card- und Terminalkomponente wird ein eigener STEP, d.h. eine ASM-Regel, erzeugt. Dieser ist nachfolgend beispielhaft anhand des Schrittes für die Smart Card-Komponente `Copycard` erläutert. Für alle weiteren Komponententypen erfolgt die Erzeugung des STEPs analog.

```
procedures
COPYCARDSTEP :
   agent × (agent → Secret) × (nat → Nonce) ×
   connections : bool × (agent → int) ×
  (agent → Nonce) × (agent → StateCard) × nat ×
  (agent → ports → messagelist) × attackerdataset;
```

Die Regel, die einen Schritt für eine Smart Card- oder Terminalkomponente durchführt, hat als Parameter die Variablen bzw. die dynamischen Funktionen, die aus den Attributen und Assoziationsenden des ausgewählten Agententyps erzeugt worden sind. Zusätzlich hat die Regel die Parameter `ag` (vom Typ `agent` für den gewählten Agenten), `all-nonces` (vom Typ `nat` → Nonce, für die Generierung einer neuen Nonce), `connections` (vom Typ `connections`, die aktuell aufgebauten Verbindungen), das Flag `exception_occurred` (vom Typ `bool`, für die Fehlerbehandlung (siehe Seite 224, Punkt 2)), `next-nonce` (vom Typ `nat`, für die Generierung einer neuen Nonce), `inputs` (vom Typ `agent` → `ports` → `messagelist`, die Inboxen) sowie `attacker-known` (vom Typ `attackerdataset`, das Angreiferwissen). Listing 8.9 zeigt die ASM-Regel COPYCARDSTEP.

Die Regel wählt zunächst nichtdeterministisch einen Port (des Agenten `ag`) aus, der gültig ist (`is-valid-port(ag,port)` gibt genau dann `true` zurück, wenn `endpoint-ok(ag` ⊙ `port) true` zurückgibt) und dessen Inbox nicht leer ist (Zeile 6-7). Anschließend wird die erste Nachricht in der zu dem Port gehörenden Inbox in der Variablen `inmsg` gespeichert (Zeile 8). Das ist die Nachricht, die in diesem Schritt verarbeitet wird. Die Nachricht wird dann aus der Inbox gelöscht (Zeile 9) und geprüft, ob es sich um eine gültige Nachricht für eine Kartenkomponente handelt (Zeile 10). Dieser Test ist nur für Steps auf Kartenseite vorhanden. Das Prädikat `is_valid_card_message` prüft, dass die als Argument übergebene Nachricht keine Benutzernachricht ist (diese können von der Karte nicht verarbeitet werden) und dass alle Werte vom Typ Integer, die in der Nachricht enthalten sind, klein genug sind, um sie als

```
1  declaration
2  COPYCARDSTEP (ag, Copycard-passphrase, all-nonces, connections
3                ; exception_occurred, Copycard-balance,
4                  Copycard-challenge, Copycard-statecard,
5                  next-nonce, inputs, attacker-known){
6  choose port
7  with (is-valid-port(ag, port) ∧ inputs(ag)(port)≠[])
8  in let inmsg = inputs(ag)(port) .first
9     in {inputs := rem(ag, port, inputs);
10        if ¬is_valid_card_message(inmsg)
11        then skip
12        else if isRequestBalance(inmsg)
13             then RequestBalance
14             else if isPay(inmsg)
15                  then Pay
16                  else if isAuthenticate(inmsg)
17                       then Authenticate
18                       else if isLoad(inmsg)
19                            then Load
20                            else STOPSTEP(; exception_occurred)}
21 ifnone skip};
```

Listing 8.9: Regel COPYCARSSTEP

short-Werte (ohne Auftreten eines Over-/Underflows) zu speichern. Schlägt der Test fehl, wird die Nachricht verworfen und die Regel COPYCARDSTEP beendet (Zeile 11). Dieser Test ist notwendig, weil die Terminals in der Implementierung Integer verwenden und verschicken, auf der Karte aber statt der nicht unterstützten Integer der primitive Typ `short` verwendet wird. In der Implementierung wird deshalb bei der Deserialisierung eines Integers geprüft, ob der Wert als `short` gespeichert werden kann und ggf. eine Exception geworfen (siehe Abschnitt 6.2.4.2, Seite 145). Ist die Nachricht `inmsg` eine gültige Kartennachricht, wird je nach Typ der Nachricht eine ASM-Regel aufgerufen, die den eigentlichen Protokollschritt durchführt, d.h. die Nachricht `inmsg` verarbeitet (Zeilen 12-19). Hat die Nachricht einen Typ, den der gewählte Agent (vom Typ `Copycard`) nicht verarbeiten kann, wird die Prozedur STOPSTEP aufgerufen. Dies bedeutet, dass eine empfangene Nachricht, die von einer Kopierkarte nicht verarbeitet werden kann, ignoriert wird.

An dieser Stelle bleibt noch zu erläutern, wie ein UML-Aktivitätsdiagramm in das formale Modell übersetzt wird. Ein Aktivitätsdiagramm modelliert ein Protokoll, das aus mehreren Schritten besteht. Durch die Zuordnung eines Protokollschrittes zu einer Partition im Aktivitätsdiagramm ist definiert, von welchem Komponententyp bzw. Agententyp der Protokollschritt ausgeführt werden kann. Die Übersetzung eines in einem Aktivitätsdiagramm modellierten Protokolls lässt sich zerlegen in die Übersetzung der einzelnen Protokollschritte. Dabei wird anhand der Partition, in der der Protokollschritt modelliert ist, bestimmt, welcher Komponententyp den Schritt durchführen kann. Hieraus wird die Information gewonnen, dass z.B. die Kopierkarte die Nachrichtentypen `RequestBalance`, `Pay`, `Authenticate` und `Load` empfangen und die entsprechenden Schritte durchführen kann.

Ein Protokollschritt wird ebenfalls in eine ASM-Regel übersetzt. Für den COPYCARDSTEP sind dies die Regeln `RequestBalance`, `Pay`, `Authenticate` und `Load`, die die gleichnamigen Nachrichten(typen) verarbeiten. Die Regel `RequestBalance` ist als Beispiel nachfolgend angegeben. Die Parameter der Regel ergeben sich aus den Parametern der aufrufenden Regel (in diesem Fall COPYCARDSTEP). Die Regel, die einen Protokollschritt durchführt, besitzt Parameter für genau die dynamischen Funktionen des gewählten Agententyps (sowie einige weitere benötigte Variablen), die in der Regel verwendet werden.

```
procedures
RequestBalance : agent × (agent → int) × connections :
                 (agent → StateCard) ×
                 (agent → ports → messagelist) × attackerdataset;

declaration
RequestBalance(ag, Copycard-balance, connections;
               Copycard-statecard, inputs, attacker-known){
Copycard-statecard(ag) := IDLE_CARD;
choose port
with (  (  port = Copycard2DepositMachineDefaultPort
         ∨ port = Copycard2CopyingMachineDefaultPort)
      ∧ connected(ag × port, connections))
in SEND(mkResRequestBalance(Copycard-balance(ag)),
            port, ag, connections; attacker-known, inputs)
ifnone skip};
```

Die Regel `RequestBalance` ändert zunächst den Zustand der Komponente auf IDLE und sendet dann eine `ResRequestBalance`-Nachricht an das Terminal zurück. Der entsprechende ASM-Code für die Zuweisung sowie den Aufruf der Prozedur SEND zum Senden einer Nachricht werden aus den im Aktivitätsdiagramm verwendeten UML-Elementen und den darin enthaltenen MEL-Ausdrücken generiert.

8.4.3.2. Transformation eines Aktivitätsdiagramms

Die Übersetzung eines in einem Aktivitätsdiagramm modellierten Protokollschritts ergibt sich aus der Übersetzung der für die Modellierung verwendeten UML-Elemente sowie der MEL-Ausdrücke, die in den UML-Elementen verwendet werden. Die Übersetzungsregeln für die UML-Elemente sind nachfolgend beschrieben. Anschließend wird die Übersetzung der MEL-Ausdrücke beschrieben.

Übersetzung der UML-Elemente Für jedes verwendete UML-Element ist angegeben, wie das Element im formalen Modell abgebildet wird.

- Ein UML-`ControlFlow` dient der Strukturierung und bestimmt die Reihenfolge, in der die UML-Elemente in ASM-Code übersetzt werden. Ein `ControlFlow` definiert somit die Hintereinanderausführung von UML-Elementen. Aus den `ControlFlow`-Elementen selber wird jedoch kein Code erzeugt.

- Eine UML-`AcceptEventAction` modelliert das Empfangen einer Nachricht. Diese entspricht im formalen Modell der Entnahme dieser Nachricht aus der entsprechenden Inbox. Es müssen dabei zwei Fälle unterschieden werden: Der Empfang einer Nachricht durch einen Benutzer und der Empfang durch eine Smart Card- oder Terminalkomponente. Empfängt ein Benutzer eine Nachricht, endet der Protokollschritt mit dem Empfang. Dies entspricht in der Realität dem Fall, dass der Benutzer eine Information vom Terminal erhält (z.B. durch grafische Anzeige der Information über eine Benutzeroberfläche). Das Lesen einer Nachricht, die sich in der Inbox eines Benutzers befindet, wurde bereits bei der Beschreibung des USER-Schrittes erläutert. Der Empfang einer Nachricht durch eine Smart Card- oder Terminalkomponente hingegen hat die Durchführung eines Protokollschrittes zur Folge. Die UML-`AcceptEventAction` selber wird nicht in ASM-Code übersetzt. Der darin enthaltene MEL-Ausdruck wird jedoch verwendet, um für die Parameter der Nachricht lokale Variablen zu deklarieren, die die Daten der Nachricht enthalten und innerhalb des Protokollschritts verwendet werden können. Dies ist bei der Erläuterung der Übersetzung der MEL-Ausdrücke beschrieben.
- Eine UML-`SendSignalAction` modelliert das Senden einer Nachricht an eine andere Komponente. Der Empfänger ergibt sich durch die Partition, in der sich die zugehörige `AcceptEventAction` befindet. Eine UML-`SendSignalAction` wird in einen Aufruf der Prozedur SEND übersetzt. Diese ist unabhängig von der konkreten Nachricht und in Listing 8.10 abgebildet.

```
1  procedures
2  SEND : message × ports × agent × connections :
3           attackerdataset × (agent → ports → messagelist);
4
5  declaration
6  SEND(outmsg, outport,ag,connections;attacker-known,inputs){
7  choose conn
8  with(is-endpoint(ag ⊙ outport, conn) ∧ conn ∈ connections)
9  in {let rem-agent = other-endpoint(ag ⊙ outport,conn).agent,
10          rem-port = other-endpoint(ag ⊙ outport, conn) .port
11     in { if attacker-can-read(conn)
12          then attacker-known := attacker-known ∫+ outmsg ;
13               inputs:=add(rem-agent,rem-port,outmsg,inputs)
14        }
15    }
16 ifnone skip};
```

Listing 8.10: Prozedur `SEND` für das Versenden einer Nachricht

Die Prozedur SEND erwartet als Argumente die zu versendende Nachricht `outmsg`, den Port `outport`, über den die Nachricht verschickt werden soll, den Sender `ag` sowie die bestehenden Verbindungen `connections`, die Inboxen `inputs` sowie das Angreiferwissen `attacker-known`. Die beiden letztgenannten sind Ein-/Ausgabeparameter, alle anderen sind reine Eingabeparameter. Die Übersetzung einer MEL-`SendSignalAction` ist bei der Beschreibung der Übersetzung der MEL-Ausdrücke erläutert. Die Prozedur SEND wählt nichtdeterministisch eine Verbindung, die aktuell aufgebaut ist und deren Endpunkt `ag ⊙ outport` ist, aus (Zeile 7-8). In

den lokalen Variablen `rem-agent` und `rem-port` wird dann der Empfängeragent und -port gespeichert (Zeile 9-10). Wenn der Angreifer die Fähigkeit hat, Nachrichten zu lesen, die über die gewählte Verbindung gesendet werden, wird diese Nachricht (sowie alle Informationen, die aus dieser Nachricht sowie dem bisherigen Angreiferwissen abgeleitet werden können) in das Angreiferwissen aufgenommen (Zeile 11-12) und die zu sendende Nachricht wird in die entsprechende Inbox hinzugefügt, d.h. hinten angehängt (Zeile 13). Gibt es keine bestehende Verbindung, über die die Nachricht gesendet werden kann, tut die Prozedur nichts (Zeile 16).

- Eine UML-`Action` enthält immer einen MEL-Ausdruck. Für die UML-`Action` selber wird kein ASM-Code erzeugt. Lediglich der in der `Action` enthaltene MEL-Ausdruck wird übersetzt.
- Ein UML-`DecisionNode` modelliert eine Verzweigung und hat zwei ausgehende `ControlFlows`, deren `Guards` MEL-Ausdrücke enthalten, die zu einem booleschen Wert ausgewertet werden (siehe Abschnitt 4.4.2.2). Eine Verzweigung in UML wird in eine Verzweigung im formalen Modell übersetzt.

```
if(guard1)
then{...}
else{...}
```

  `guard1` ist einer der Guards der beiden `ControlFlows` (er wird bei der Transformation „zufällig“ gewählt). Eine Voraussetzung an die Modellierung ist, dass beide Guards zu komplementären Werten ausgewertet werden. Aus diesem Grund ist es egal, welcher Guard für den Test der Fallunterscheidung gewählt wird. Enthält einer der beiden Guards das Schlüsselwort `else`, wird der andere Guard für den Test verwendet. Ein UML-`DecisionNode` führt dazu, dass zwei parallele Kontrollflüsse entstehen. In der Regel endet einer der beiden mit einem `FlowFinalNode`, d.h. modelliert das Auftreten eines Fehlers, und der andere mit einem `ActivityFinalNode`, d.h. modelliert das Ende des Protokolls. In diesem Fall werden die beiden Kontrollflüsse nicht mehr zusammengeführt, sondern der `then` sowie der `else`-Block enden erst mit Erreichen der `FinalNodes`. Eine andere Möglichkeit ist, beide Kontrollflüsse durch einen UML-`MergeNode` wieder zusammenzuführen.
- Ein UML-`MergeNode` wird verwendet, um verschiedene Kontrollflüsse wieder zusammenzuführen. Einem `MergeNode` ist (mindestens) ein `DecisionNode` vorausgegangen, durch den mehrere Kontrollflüsse entstanden sind. Bei der Generierung der ASM (sowie auch der Generierung des Codes) bewirkt ein UML-`Merge Node`, dass der ASM-Code, der für die UML-Elemente hinter dem `MergeNode` generiert wird, für beide Kontrollflüsse dupliziert wird. Technisch gesehen findet somit keine Zusammenführung der Kontrollflüsse statt.
- Ein UML-`InitialNode` beschreibt den Startpunkt eines Protokolls und somit auch des ersten Protokollschritts. Protokollläufe können im SecureMDD-Ansatz nur von Benutzern begonnen werden, d.h. diese beginnen einen neuen Lauf, indem sie eine Benutzernachricht an ein Terminal senden. Da dies über die (anwendungsunabhängige) ASM-Regel USERSTEP spezifiziert ist, muss der UML-`InitialNode` nicht in das formale Modell übersetzt werden. Er dient nur der Strukturierung und Visualisierung des Beginns eines Protokolls.

- Ein UML-`ActivityFinalNode` beschreibt das Ende eines Protokollschritts, in dem keine Nachricht an eine andere Komponente gesendet werden soll. Ein `ActivityFinalNode` beendet neben dem Protokollschritt auch ein Protokoll. Bei Erreichen eines `ActivityFinalNodes` ist nichts mehr zu tun. Dieser UML-Knoten wird deshalb in ein leeres „Codefragment“ übersetzt.
- Ein UML-`FlowFinalNode` modelliert das Auftreten eines expliziten Fehlers. Dieser führt auf Implementierungsebene zum Werfen einer Exception und dem Beenden des Protokollschritts. Im formalen Modell muss dieses Verhalten adäquat abgebildet werden. Ein UML-`FlowFinalNode` wird deshalb in den Aufruf der Prozedur STOPSTEP übersetzt.

  ```
  procedures
  STOPSTEP : bool;

  declaration
  STOPSTEP (;exception_occurred) {
     exception_occurred := true };
  ```

  Diese setzt die boolesche Variable `exception_occurred` auf `true`. Da der `FlowFinalNode` keine ausgehenden `ControlFlows` besitzt, ist damit der aktuelle Protokollschritt beendet. Das Flag `exception_occurred` ist nur für den Fall relevant, dass der Fehler innerhalb eines Subdiagramms auftritt (siehe Seite 224).

Subdiagramme sind im formalen Modell als Prozeduren deklariert, die den gleichen Namen wie das Diagramm haben. Die Ein- und Ausgabeparameter ergeben sich aus den UML-`ActivityParameterNodes` des Diagramms sowie den dynamischen Funktionen der Komponente, für die das Subdiagramm definiert ist. Der Grund hierfür ist, dass innerhalb eines Subdiagramms sowohl auf die als Argumente übergebenen Werte als auch auf die dynamischen Funktionen der Komponente zugegriffen werden kann. Außerdem besitzt jedes Subdiagramm Parameter für den Agenten `ag`, der das Subdiagramm aufgerufen hat sowie das boolesche Flag `exception_occurred`, das angibt, ob innerhalb des Subdiagramms ein Fehler aufgetreten ist. Innerhalb des Subdiagramms werden dieselben UML-Elemente verwendet wie in einem Aktivitätsdiagramm, das ein Protokoll modelliert. Die Generierung folgt analog zu der Generierung der Protokollschritte. Zusätzlich werden in Subdiagrammen UML-`JoinNodes` verwendet, um die von den `ActivityParameterNodes`, die die Eingabeparameter definieren, ausgehenden Kontrollflüsse zu einem Kontrollfluss zusammenzuführen. Im formalen Modell werden die `ActivityParameterNodes` jedoch in Parameter der Prozedur übersetzt. Eine Übersetzung des UML-`JoinNodes` in Code ist deshalb nicht notwendig.

Übersetzung der MEL-Ausdrücke Die Übersetzung der MEL-Ausdrücke in eine formale Spezifikation, genauer in die Rümpfe der ASM- und Subdiagrammprozeduren ist an vielen Stellen intuitiv und birgt keine Überraschungen (z.B. die Deklaration von lokalen Variablen oder das Erzeugen eines neuen Objekts bzw. Element eines Datentyps). Die Semantik der in den UML-Elementen verwendeten Ausdrücke der Sprache MEL wurde bereits in Abschnitt 5.2 beschrieben. Aus diesem Grund wird die Übersetzung der MEL-Ausdrücke in das formale Modell im Folgenden nur für die Teile beschrieben, bei denen diese nicht offensichtlich klar ist. Diese sind im Folgenden aufgelistet und werden anschließend erläutert:

1. Übersetzung der binären, arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division und Modulo)
2. Übersetzung des Auftretens eines Fehlers (innerhalb eines Subdiagramms)
3. Übersetzung einer MEL-`AcceptEventAction`
4. Übersetzung einer MEL-`SendSignalAction`
5. Übersetzung eines Aufrufs der MEL-Methode `decrypt`
6. Kopiersemantik im formalen Modell

1. Übersetzung der binären, arithmetischen Operationen
Auf Smart Card-Seite wird vor Ausführung einer arithmetischen Operation geprüft, dass bei Ausführung dieser kein Over- oder Underflow auftritt. Der Grund hierfür ist, dass diese Operationen auf Kartenseite auf `short`-Werten ausgeführt werden. Im formalen Modell werden statt der short-Werte Integer verwendet. Es muss jedoch auch hier, damit der Code eine Verfeinerung des formalen Modells ist, im formalen Modell überprüft werden, ob das Ergebnis der Operation in den Wertebereich eines Short-Wertes passt. Vor Ausführung einer arithmetischen Operation wird deshalb in der formalen Spezifikation mithilfe des Prädikats `shortOverflow` geprüft, ob bei Ausführung der Operation ein Over- oder Underflow stattfinden würde. In diesem Fall wird die Prozedur STOPSTEP aufgerufen und der aktuelle Protokollschritt nicht weiter ausgeführt. Anderenfalls wird die Operation ausgeführt und der weitere Protokollschritt durchgeführt. Nachfolgend ist beispielhaft die Übersetzung des Reduzierens des Kontostands auf der Kopierkarte dargestellt. Dies entspricht dem MEL-Ausdruck `balance := balance - value` im UML-Aktivitätsdiagramm zum Bezahlen von Kopien.

```
if (shortOverflow(Copycard-balance(ag) - value))
then STOPSTEP(; exception_occurred)
else{
  Copycard-balance(ag) := Copycard-balance(ag) - value;
  ...
}
```

2. Übersetzung des Auftretens eines Fehlers innerhalb eines Subdiagramms
Der Aufruf eines Subdiagramms wird im formalen Modell in den Aufruf der gleichnamigen Prozedur übersetzt. Bei Aufruf eines Subdiagramms ist zu beachten, dass das Auftreten eines Fehlers während der Ausführung der Subdiagramm-Prozedur an den Aufrufer weitergegeben werden muss. Der Aufruf geschieht immer von einer Prozedur aus, die einen Protokollschritt durchführt. Tritt innerhalb des Subdiagramms ein Fehler auf, darf die aufrufende Prozedur die Abarbeitung des Protokollschritts nicht fortsetzen, sondern muss diesen beenden. Die Prozedur STOPSTEP, die bei Auftreten eines Fehlers aufgerufen wird, setzt deshalb die boolesche Variable `exception_occurred` auf `true`.

Nachdem das Subdiagramm ausgeführt wurde, prüft die ASM, ob die Variable gleich `true` ist und beendet in diesem Fall den Protokollschritt (`skip`). Anderenfalls wird der Protokollschritt weiter ausgeführt. Dies ist in nachfolgendem Listing abgebildet.

```
SUBDIAGRAMMAUFRUF(...,exception_occurred)
if (exception_occurred)
```

```
then {skip}
else {...}
```

3. Übersetzung einer MEL-`AcceptEventAction`
Eine MEL-`AcceptEventAction` modelliert das Empfangen einer Nachricht. Eine `AcceptEventAction` enthält Namen für die lokalen Variablen, in denen beim Empfang die in der Nachricht enthaltenen Daten gespeichert werden. Eine `AcceptEventAction` wird in die Deklaration lokaler Variablen für die Argumente der empfangenen Nachricht übersetzt. Die Deklaration geschieht in der ASM-Regel, die die Nachricht verarbeitet. Für die MEL-`AcceptEventAction Load(value,hashedauth)` der Kopierkartenanwendung werden in der Regel `Load` die lokalen Variablen `value` und `hashedauth` deklariert. Im Anschluss wird der Protokollschritt durchgeführt. Dies ist beispielhaft nachfolgend abgebildet.

```
Load(..){
let value = inmsg .amount, hashedauth = inmsg .authterminal
in {..}}
```

4. Übersetzung einer MEL-`SendSignalAction`
Eine MEL-`SendSignalAction` modelliert das Senden einer Nachricht. Im formalen Modell entspricht das dem Schreiben der Nachricht in die Inbox des Empfängers. Durch die MEL-`SendSignal Action` ist modelliert, welche Nachricht verschickt werden soll. Diese wird mithilfe des Konstruktors für die Nachricht neu erzeugt und anschließend beim Aufruf der Prozedur SEND als Argument (`outmsg`) übergeben. Die MEL-`SendSignalAction Load(amountToLoad, hashedauth)` (die den Wert des Attributs `amountToLoad` der DepositMachine und den in der lokalen Variablen `hashedauth` gespeicherten Hashwert versendet) wird im formalen Modell in den Aufruf

```
SEND(mkLoad(DepositMachine-amountToLoad(ag), hashedauth),
     DepositMachine2CopycardDefaultPort, ag, connections;
     attacker-known, inputs)
```

übersetzt. Das erste Argument ist das neu erzeugt Element vom Typ `Load`. Als zweites Argument muss der Port, über den die Nachricht verschickt werden soll, angegeben werden. Da es zwischen der `DepositMachine` (dem Sender) und der `Copycard` (dem Empfänger, beide ergeben sich aus den Partitionen des zugehörigen Aktivitätsdiagramms) im Deploymentdiagramm nur einen Kommunikationspfad gibt, wird beim Senden der `Load`-Nachricht der für diese Verbindung generierte Defaultport `DepositMachine2CopycardDefaultPort` verwendet. Ist die Zuordnung nicht eindeutig (siehe Abschnitt 4.2.2.1), muss in der `SendSignalAction` über das Schlüsselwort `via` der Name des (im Deploymentdiagramm definierten) Ports angegeben werden, über den die Nachricht gesendet werden soll. Dieser Name wird dann bei der Generierung des Aufrufs der Prozedur SEND für den Parameter `outport` verwendet.

5. Übersetzung eines Aufrufs der MEL-Methode `decrypt`
Die MEL-Methode `decrypt` zum Entschlüsseln eines symmetrisch oder asymmetrisch verschlüsselten Dokuments wird bei Generierung der formalen Spezifikation in eine gleichnamige Funktion übersetzt (siehe Abschnitt 8.4.1, Seite 195). Nachfolgend ist der Aufruf der Funktion `decrypt` in der ASM dargestellt.

```
if ¬ (can_decrypt(privkey,encdataasymm)
```

```
        ∧ isDataClass(decrypt(privkey,encdataasymm)))
then STOPSTEP(; exception_occurred)
else ...
```

In dem Beispiel wurde ein Element vom Typ `DataClass` mit einem öffentlichen Schlüssel asymmetrisch verschlüsselt, d.h. im UML-Modell ist die Datenklasse `DataClass` mit Stereotyp ≪PlainData≫ annotiert. `privkey` ist eine Variable vom Typ `PrivateKey`, `encdataasymm` ist ein Element vom Typ `EncDataAsymm` (d.h. ein asymmetrisch verschlüsseltes Dokument). Die Funktion `decrypt` wird nur aufgerufen, wenn das Prädikat `can_decrypt true` zurückgibt (d.h. der private Schlüssel `privkey` passt zu dem öffentlichen Schlüssel, mit dem verschlüsselt wurde) und das entschlüsselte Element den richtigen Typ hat (dies prüft das Prädikat `isDataClass`). Sind diese Voraussetzungen nicht erfüllt, wird die Funktion `decrypt` nicht aufgerufen und der aktuelle ASM-Schritt bricht durch Aufruf der Prozedur STOPSTEP ab. Anderenfalls wird die Funktion `decrypt`, die das verschlüsselte Dokument entschlüsselt, aufgerufen.

6. Kopiersemantik im formalen Modell
Wie auch die Model Extension Language besitzt das formale Modell eine Kopiersemantik. Dies bedeutet, dass die abstrakten Datentypen, auf deren Elementen die ASM operiert, keine Identitäten für die Daten verwenden. Die algebraisch spezifizierten Daten sind gleich, wenn ihre Werte gleich sind.

8.5. Verwandte Arbeiten

Zur Verifikation von Sicherheitseigenschaften für kryptographische Protokolle gibt es im Wesentlichen drei unterschiedliche Ansätze. Dies sind zunächst manuelle, papierbasierte Verfahren. Um das Risiko von Fehlern in den erstellten Beweisen zu reduzieren, werden inzwischen allerdings hauptsächlich computergestützte Beweissysteme verwendet. Dabei kommen zwei unterschiedliche Techniken zum Einsatz. Während Model Checker versuchen, Eigenschaften nach der Formalisierung vollautomatisch zu beweisen, verlangen interaktive Verifikationsansätze auch während des Beweisens Eingaben durch den Verifikateur.

Alle drei Ansätze werden im Folgenden kurz vorgestellt. Außerdem gibt es eine Reihe weiterer Arbeiten zur Verifikation von kryptographischen Protokollen, die nicht in eine dieser drei Kategorien fallen. Dies sind zum Beispiel Strand Spaces, die von dem Tool Athena unterstützt werden [176] oder der NRL Protocol Analyzer von Meadows [123].

Eine ausführliche Beschreibung (der meisten) der hier genannten Arbeiten findet sich in [78].

8.5.1. Manuelle Beweise

Die Bandbreite manueller Sicherheitsbeweise über kryptographischen Protokollen ist sehr groß. Da diese Ansätze in der Regel schlecht zwischen Fallstudien skalieren und wenig verbreitet sind, sollen hier nur die Arbeit von Bella und Riccobene [18,19] sowie die von Rosenzweig et. al. [163] genannt werden. Bella und Riccobene haben in einem ASM-basierten Verfahren ohne Rechnerunterstützung [18] zunächst die Korrektheit des Kerberos Protokolls gezeigt. In der Folge haben die Forscher ihren Ansatz verallgemeinert [19]. Auch Rosenzweig et. al.

stellen in [163] einen ASM-basierten Ansatz zur Modellierung und einen papierbasierten Korrektheitsbeweis für kryptographische Protokolle vor.

8.5.2. Model Checking

Model-Checking ist eine grundsätzlich sehr effektive Methode, um Fehler in Protokollen zu finden. So ist der 1995 von Lowe mit Model-Checking gefundene und in [117] beschriebene Man-In-The-Middle Angriff auf das Needham-Schroeder-Authentifizierungsprotokoll [143] noch heute eines der prominentesten und meistverwendeten Beispiele für die Notwendigkeit von Sicherheitsanalysen.

Allen Model-Checkern ist gemein, dass sie aus der Verhaltensbeschreibung der Protokollteilnehmer und des Angreifers den erreichbaren Zustandsraum aufbauen. Der Nachteil des Model-Checkings ist, dass der Zustandsraum des Systems endlich sein muss. Bei den Anwendungen, die in SecureMDD üblicherweise betrachtet werden, ist der Zustandsraum in der Regel jedoch nicht endlich. Der Grund hierfür ist, dass die Anzahl der an einer Anwendung beteiligten Komponenten in der Regel nicht fixiert ist, d.h. es gibt beliebig viele Komponenteninstanzen (z.B. Kopierkarten). Außerdem kann es sein, dass einige Angriffe erst durch die Betrachtung mehrerer Protokollläufe entstehen können und somit eine prinzipiell unbeschränkte Zahl an Protokollläufen zu berücksichtigen ist. Model-Checker versuchen durch verschiedene Techniken den Zustandsraum einzuschränken. In der Regel geschieht dies durch Abstraktion des Anwendungsmodells sowie der Begrenzung der beteiligten Komponenteninstanzen und Protokollläufe auf eine fixe Anzahl. Einen Vergleich der Leistungsfähigkeit unterschiedlicher Model-Checker führt [44] durch.

Mittlerweile gibt es eine große Zahl unterschiedlicher Model-Checker, die für die Protokollverifikation entwickelt werden. Dazu gehören beispielsweise Scyther [45] oder ProVerif [22]. AVISPA [8] ist eine Toolsuite, die verschiedene Model-Checker integriert. Daneben existieren Ansätze, generische Model-Checker auch im Kontext von Sicherheitsprotokollen einzusetzen, beispielsweise [94] oder [119].

Üblicherweise werden Model-Checker zu Verifiaktion von Standardsicherheitseigenschaften verwendet. Bislang ist kein Model-Checking-Ansatz bekannt, der es ermöglicht, beliebige applikationsspezifische Sicherheitseigenschaften zu beweisen.

8.5.3. Interaktive Verifikation

Interaktive Verifikation beschreibt zusammenfassend eine Reihe möglicher Techniken, bei denen der Verifikateur Einfluss auf den eigentlichen Beweisvorgang nimmt. Üblicherweise führt das System dabei eine Reihe automatischer Vereinfachungen durch und stoppt die automatische Bearbeitung immer dann, wenn schwer automatisierbare Schritte wie Generalisierungen oder Invariantendefinitionen notwendig sind.

Im Gegensatz zum Model-Checking ist es nicht notwendig, den Zustandsraum des Systems vor Start der Verifikation zu begrenzen. Stattdessen können Invarianten- und Induktionstechniken verwendet werden, um den Zustandsraum auf endlich viele zu betrachtende Zustandsgruppen zu begrenzen, die dann nacheinander evaluiert werden. Dadurch ist es im Gegensatz zum Model-Checking grundsätzlich möglich, komplexe domänenspezifische Aussagen zu formulieren und zu beweisen, sofern der konkrete Ansatz dies vorsieht.

Paulson stellt in [17] seinen *Inductive Approach* vor. Dieser dient der Formalisierung von Sicherheitsprotokollen und der Einbettung der Formalisierung in das Verifikationssystem Isabelle/HOL [146]. Mit diesem Ansatz konnten erfolgreich diverse Protokolle wie TLS [153], Kerberos [16] oder SET (Secure Electronic Transactions) [15] untersucht werden.

Schneider [170] verwendet die Temporallogik CSP um Sicherheitsprotokolle zu formalisieren. Das formale Modell wird in das interaktive Verifikationssystem PVS [151] eingebettet und dort verifiziert. Der Ansatz wird in [164] detailliert vorgestellt.

In [121] verwenden McCarthy et al. den interaktiven Theorembeweiser Coq, um Aussagen für kryptographische Protokolle zu verifizieren. Corbineau et al. [41] betrachten ebenfalls die Sicherheit kryptographischer Protokolle. Die gemachten Beweise werden mit Coq geführt. Betrachtet wird ein Angreifer, der nicht nur eine „Black-Box“-Sicht auf das System, sondern zusätzlich auch direkten Zugriff auf die Komponenten (Seitenkanalangriffe) hat.

9 Beweistechnik

Zusammenfassung: Nachdem die formale Spezifikation aus dem plattformunabhängigen UML-Modell erzeugt wurde, kann sie automatisch in das interaktive Beweissystem KIV [9] eingelesen werden. Mithilfe des Tools ist die interaktive Verifikation sowohl von anwendungsspezifischen Sicherheitseigenschaften als auch von anwendungsunabhängigen Eigenschaften, wie die Geheimhaltung oder Integrität von Daten, für die modellierte Anwendung möglich. Dieses Kapitel erläutert, wie sich Sicherheitseigenschaften ausdrücken und als entsprechende Theoreme und Invarianten formalisieren lassen. Weiterhin beschreibt dieses Kapitel die für die Verifikation verwendete Beweistechnik anhand des Beispiels der Kopierkarte. Die beschriebenen Ergebnisse sind in [136] publiziert.

Für die Kopierkartenanwendung wird die Sicherheitseigenschaft betrachtet, dass der Betreiber der Kopierkarten kein Geld verliert. Was man also garantieren möchte ist, dass zu jedem Zeitpunkt die Summe des Betrags, der in die Ladeterminals eingeworfen wurde, immer größer oder gleich des Betrags ist, der von den Karten wieder abgebucht wurde und für den Kopien angefertigt wurden. Man möchte also sicherstellen, dass es nicht möglich ist, mehr Geld am Kopiergerät auszugeben als vorher mithilfe eines echten Terminals auf eine Karte geladen wurde. Fertigt ein Betrüger mit einer gefälschten Kopierkarte Kopien an, ist diese Eigenschaft verletzt. Wird die Karte „illegal“, d.h. an einem Heim-PC, aufgeladen und anschließend erfolgreich zum Bezahlen von Kopien verwendet, ist diese Eigenschaft ebenfalls verletzt.

Das Kapitel gliedert sich wie folgt. In Abschnitt 9.1 wird die Verwendung von Ghostvariablen beschrieben und illustriert, wie die oben genannte Sicherheitseigenschaft mithilfe von Ghostvariablen ausgedrückt werden kann. Anschließend wird in Abschnitt 9.2 die verwendete Beweistechnik anhand des Beispiels der Kopierkartenanwendung erläutert.

9.1. Verwendung von Ghostvariablen im plattformunabhängigen UML-Modell

Um sicherstellen, dass eine Sicherheitseigenschaft für eine modellierte Anwendung gilt, muss man in der Lage sein, diese im Beweissystem auszudrücken. Oft benötigt man hierfür Hilfsvariablen, auch Ghostvariablen genannt. Ghostvariablen sind im plattformunabhängigen UML-

Modell mit dem Stereotyp ≪Ghostvariable≫ annotiert und werden für die Modellierung der eigentlichen Funktionalität der Anwendung nicht benötigt. Sie dienen einzig und allein dazu, die gewünschten Sicherheitseigenschaften auszudrücken. Der Stereotyp ≪Ghostvariable≫ erweitert die Metaklasse `Property` und kann somit auf Attribute und Assoziationsenden angewendet werden (siehe Abschnitt 4.1.3, Seite 47).

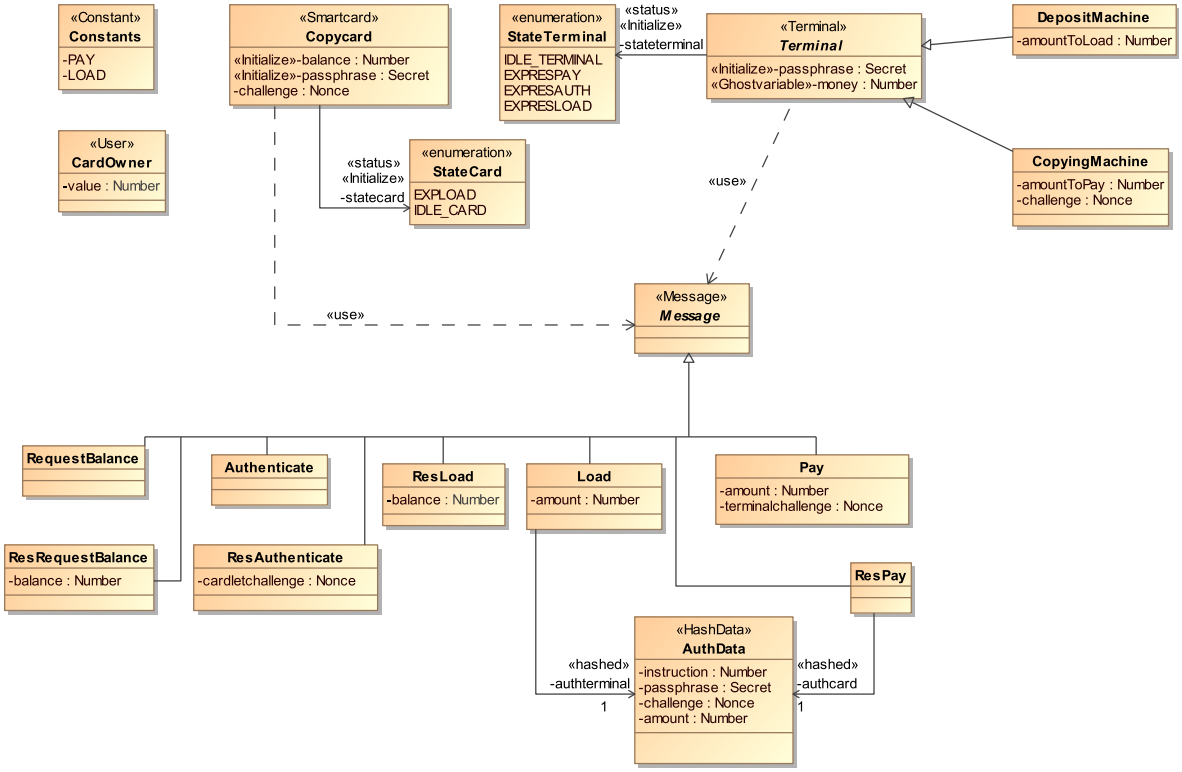

Abbildung 9.1.: Klassendiagramm der Kopierkartenanwendung mit Ghostvariable `money`

Im Folgenden wird nun die oben genannte Eigenschaft für die Kopierkartenanwendung mithilfe von Ghostvariablen beschrieben. Diese besagt, dass „zu jedem Zeitpunkt die Summe des Betrags, der in die Ladeterminals eingeworfen wurde, immer größer oder gleich des Betrags ist, der von den echten Kopierkarten wieder abgebucht wurde“.

Es soll also etwas darüber ausgesagt werden, wie viel Geld in die Ladeterminals eingeworfen und für wie viel Geld Kopien angefertigt wurden. Um dies zu zählen, wird eine Ghostvariable `money` verwendet. Diese wird als Attribut in der abstrakten Klasse `Terminal` des plattformunabhängigen UML-Modells der Anwendung (siehe Abschnitt 4.3) hinzugefügt. Abbildung 9.1 zeigt das Klassendiagramm, in dem die abstrakte Klasse `Terminal` um das Attribut `money` erweitert wurde. Jedes Ladeterminal sowie Kopiergerät hat somit ein Attribut `money`. Für ein Ladeterminal zählt dieses Attribut, wie viel Geld bis zum aktuellen Zeitpunkt in dieses Terminal eingeworfen wurde. Für ein Kopiergerät zählt dieses Attribut, wie viel Geld von (echten) Kopierkarten abgebucht wurde, um dann Kopien anzufertigen.

Die entsprechende Sicherheitseigenschaft lautet dann, dass die Summe der `money`-Attribute aller `DepositMachine`- und `CopyingMachine`-Instanzen immer größer oder gleich 0 sein muss.

Was nun noch fehlt ist das Erhöhen bzw. Reduzieren des Attributs `money`, wenn Geld in ein Terminal eingeworfen bzw. von einer Kopierkarte abgebucht wird. Aus diesem Grund werden die vorgestellten Aktivitätsdiagramme für das Aufladen der Karte und das Anfertigen von Kopien aus Abschnitt 4.3 erweitert. Abbildung 9.2 (links) zeigt den ersten Protokollschritt des Ladeterminals (`UInsertMoney`). Nach dem Einwerfen des Geldes durch den Benutzer bzw. nachdem dem Terminal der aufzuladende Betrag mittels einer `UInsertMoney` Nachricht mitgeteilt wird, überprüft das Terminal, ob der übertragene Wert größer als 0 ist. Ist dies der Fall, wird der Wert der lokalen Variablen `val` im Attribut `amountToLoad` gespeichert und das `money` Feld entsprechend des Betrags erhöht. Das Attribut `money` zählt somit mit, wie viel Geld in ein Ladeterminal eingeworfen wurde. Der Rest des Ladeprotokolls bleibt unverändert.

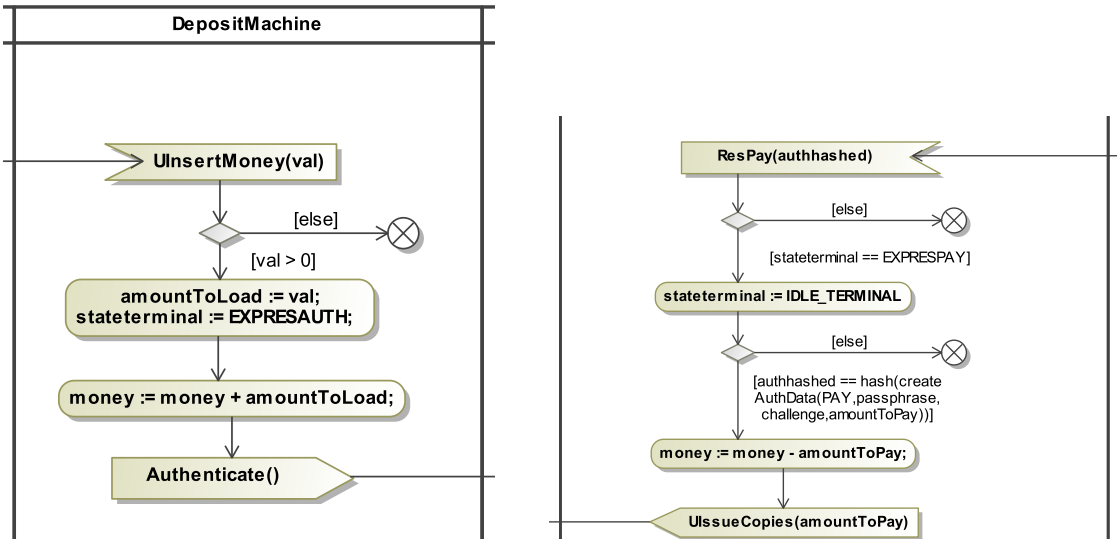

Abbildung 9.2.: Ausschnitte aus den Aktivitätsdiagrammen zum Aufladen (links) sowie zum Anfertigen von Kopien (rechts), mit Ghostvariablen

Abbildung 9.2 (rechts) zeigt das Empfangen und Verarbeiten einer `ResPay` Nachricht durch das Kopierterminal. Diesem Schritt ist das Abbuchen des für Kopien zu zahlenden Geldbetrags von der Kopierkarte vorausgegangen und dem Schritt folgt das Anfertigen der Kopien. Hier muss also der Wert des Attributs `money` entsprechend um den gezahlten Betrag `amountToPay` reduziert werden. Dies geschieht nach der Überprüfung des Zustands und des Hashwerts und vor dem Senden der `UIssueCopies` Nachricht an den Benutzer. Der Rest des Protokolls als auch alle anderen UML-Diagramme bleiben unverändert.

9.2. Verifikation von Eigenschaften

Um die informelle Sicherheitseigenschaft „der Kopiergerätebetreiber verliert kein Geld“ zu formalisieren, muss die Eigenschaft, dass die Summe der Werte der `money`-Attribute aller

Terminal-Instanzen immer größer oder gleich 0 ist, formal notiert werden. Hierfür wird zunächst eine Funktion definiert, die die Summe des Geldes, das in die Ladeterminals eingeworfen wurde, minus der Geldbeträge, die an Kopiergeräten zum Bezahlen von Kopien ausgegeben wurde, zusammenzählt. Die Axiomatisierung ist nachfolgend angegeben.

```
functions
 allMoney : (agent → int) → int;
 allMoney : (agent → int) × nat → int;

axioms
allMoney(money) =
    allMoneyDepositMachines(money, NUMOFDEPOSITMACHINES)
  + allMoneyCopyingMachines(money, NUMOFCOPYINGMACHINES);

allMoneyDepositMachines(money, 0) = 0;
allMoneyDepositMachines(money, n + 1) =
  money(DepositMachine(n)) + allMoney(money, n);

allMoneyCopyingMachines(money, 0) = 0;
allMoneyCopyingMachines(money, n + 1) =
  money(CopyingMachine(n)) + allMoney(money, n);
```

NUMOFDEPOSITMACHINES ist die Anzahl der Ladeterminals und NUMOFCOPYINGMACHINES die Anzahl der vorhandenen Kopiergeräte. Beide Konstanten haben endliche, aber beliebig wählbare Werte. Die Funktion allMoney berechnet wie viel Geld an allen existierenden Ladeterminals zusammen eingeworfen wurde und addiert die Summe des (negativen) Betrags, der an allen Kopiergeräten für Kopien ausgegeben wurde.

Die Sicherheitseigenschaft, die immer gelten soll, ist, dass diese Summe größer oder gleich 0 ist (allMoney(money) ≥ 0). Konkret heißt das, es muss gezeigt werden, dass nach Ausführen jeder möglichen Sequenz von Protokollschritten die Sicherheitseigenschaft immer noch gilt. Diese Beweisverpflichtung ist nachfolgend abgebildet.

```
init(..) ⊢ [ASM(..)] allMoney(money) ≥ 0;      (1)
```

[.] ist der Box-Operator der dynamischen Logik. $[\alpha]\varphi$ bedeutet, dass wenn das Programm α terminiert, danach die Bedingung φ gilt. Die ASM startet in einem Initialzustand (z.B. ist das Guthaben der Karten gleich 0, die Komponenten sind in ihren IDLE-Zuständen und das Angreiferwissen ist leer). Der Initialzustand ist durch ein Prädikat beschrieben, dessen Axiomatisierung nach der Generierung der formalen Spezifikation von Hand hinzugefügt werden muss (siehe Abschnitt 4.2.1.9). Anschließend werden die Protokollschritte zufällig gewählt und ausgeführt. Nach Terminierung der ASM (d.h. wenn die boolesche Variable stop auf true gesetzt wurde, siehe Abschnitt 8.4.3), muss die Sicherheitseigenschaft gelten. Dies bedeutet, es werden alle endlichen Sequenzen von Schritten (STEPs) betrachtet.

Um die Beweisverpflichtung (1) zeigen zu können, muss verifiziert werden, dass die Eigenschaft gilt, wenn sich die ASM im Initialzustand befindet, d.h.

```
init(..) ⊢ allMoney(money) ≥ 0;
```

Außerdem muss gelten, dass die Beweisverpflichtung bezüglich jedes möglichen Schrittes (STEP) der Anwendung invariant ist. Diese Beweisverpflichtung ist nachfolgend abgebildet.

```
Base-INV(..) ∧ allMoney(money) ≥ 0              (2)
⊢ [STEP(..)] allMoney(money) ≥ 0;
```

Dies bedeutet man beweist (bzw. versucht zu beweisen), dass wenn eine Basis-Invariante (Base-INV) und die Sicherheitseigenschaft gelten und der ASM-Schritt STEP terminiert, danach die Sicherheitseigenschaft nach wie vor gilt. Für die generierte ASM terminiert die ASM-Regel STEP immer.

Basis-Invariante:
Um zu beweisen, dass die Invariante (2) gilt, muss zunächst verifiziert werden, dass verschiedene andere Invarianten gelten. Diese sind zu einer Basis-Invariante (Base-INV) zusammengefasst und werden für den eigentlichen Invariantenbeweis (2) benötigt. Die Basis-Invariante umfasst anwendungsunabhängige sowie anwendungsspezifische Eigenschaften. Anwendungsunabhängige Eigenschaften machen allgemeine Aussagen, z.B. über das Wissen des Angreifers oder die Kommunikationsstruktur. Beispiele hierfür sind:

- der Angreifer kennt keine Nonces, die die Funktion `generateNonce()` in Zukunft zurückliefern kann. Konkret bedeutet dies, dass der Angreifer keine Nonces kennt, die sich in dem Pool von Nonces, die noch nicht verwendet wurden, befinden.
- Nachrichten, die aus dem Angreiferwissen abgeleitet werden können, können auch dann noch abgeleitet werden, wenn dem Wissen Daten hinzugefügt werden.
- Für alle Verbindungen, die in der Menge `connections` gespeichert sind, gilt das Prädikat `conn-ok`, d.h. diese Verbindungen sind gültig (siehe Abschnitt 8.4.3, Seite 207).

Anwendungsspezifische Eigenschaften beziehen sich auf die modellierte Anwendung. Für die Kopierkartenanwendung sind dies zum Beispiel:

- der Angreifer kennt zu keinem Zeitpunkt das Geheimnis, das von allen echten Terminals und Smart Cards zur Authentisierung verwendet wird.
- der Kontostand aller Kopierkarten ist immer größer oder gleich 0 (alle Karten werden mit Wert 0 initialisiert).
- die Variable `challenge` zweier verschiedener Agenten speichert niemals dieselbe Nonce.
- der Angreifer kann alle Nachrichten, die über Verbindungen versendet wurden, die er abhören kann, aus seinem Wissen generieren.

Außerdem gibt es anwendungsabhängige Eigenschaften, die zwar abhängig von der konkreten Anwendung sind, aber die ganz offensichtlich gelten und deshalb nicht bewiesen werden müssen. Ein Beispiel ist, dass der Angreifer die Nachrichten, die zwischen einem Terminal und einer Kopierkarte ausgetauscht werden, lesen kann. Das entsprechende Theorem lautet,

dass der Angreifer die Nachrichten lesen kann, die (am Ende eines Protokollschrittes) in die Inboxen der Terminals und Kopierkarten geschrieben werden.

Insgesamt umfasst die Basis-Invariante für die Kopierkartenanwendung ungefähr zehn Invarianten.

Für die Verifikation der Beweisverpflichtung (1) werden außerdem die Aussagen benötigt, dass die Basis-Invariante gilt, wenn die ASM im Initialzustand ist und dass die Basis-Invariante außerdem invariant über jedem möglichen Schritt (STEP) der Anwendung ist. Beide Beweisverpflichtungen sind nachfolgend aufgelistet:

```
init(..) ⊢ Base-INV(..);
Base-INV(..) ⊢ [STEP(..)] Base-INV(..);
```

Erweiterter Invariantenbeweis:
Für die allermeisten Anwendungen gilt die Beweisverpflichtung (2) leider nicht. Dies bedeutet, dass zusätzliche Invarianten benötigt werden, unter deren Verwendung dann (2) gezeigt werden kann. Welche zusätzlichen Invarianten benötigt werden, ist abhängig von der konkreten Anwendung. Im Folgenden wird dies anhand der Kopierkartenanwendung erläutert.

Die Ungleichung `allMoney(money)` $\geq$ `0` gilt zwar vor und nach jedem (vollständigen, nicht vom Angreifer beeinflussten) Protokolllauf, ist aber nicht für jeden einzelnen Protokollschritt (STEP) invariant. Empfängt zum Beispiel ein Kopiergerät eine `ResPay`-Nachricht, wird die Variable `money` um den gezahlten (und auf der Karte abgebuchten) Betrag reduziert. Die Information, dass der abgebuchte Betrag in einem früheren Protokolllauf des Ladeprotokolls auch in ein Ladeterminal eingeworfen wurde (und der Wert der `money`-Variablen des Terminals somit um diesen Betrag erhöht wurde), geht verloren, wenn man nur einen einzelnen Protokollschritt betrachtet.

Die Lösung hierfür ist, die Sicherheitseigenschaft zu erweitern und alle Informationen, die Einfluss auf die Werte der Variablen `money` haben, ebenfalls in der Sicherheitseigenschaft zu erfassen. Zum Beispiel bedeutet ein Erhöhen des Wertes einer Variablen `money` bei Einwurf eines Geldbetrags in das Ladterminal, dass dieser Betrag auch auf der Kopierkarte, die aufgeladen wird, gutgeschrieben wird (sofern der Angreifer die `Load`-Nachricht nicht unterdrückt). Das heißt, in dem auf das Einwerfen des Geldes folgenden Protokollschritt (`Load`) wird die Variable `balance` der Kopierkarte um den eingeworfenen Betrag erhöht. Wird die Variable `money` beim Bezahlen von Kopien um einen Wert reduziert, wurde in dem vorherigen Protokollschritt der Wert der Variablen `balance` der (für das Bezahlen verwendeten) Kopierkarte ebenfalls um diesen Wert reduziert. Aus diesem Grund wird der Kontostand ebenfalls in die Sicherheitseigenschaft mit aufgenommen. Analog zu der Funktion `allMoney` berechnet die Funktion `allBalance(balance)` mit der Signatur `(agent` $\rightarrow$ `int)` $\rightarrow$ `int` die Summe der Werte der `balance`-Felder aller existierenden Kopierkarten, d.h. das gesamte Guthaben aller Kopierkarten. Die erweiterte Sicherheitseigenschaften hat die Form `allMoney(money)` $\geq$ `allBalance(balance)` und umfasst somit auch die Kontostände aller Kopierkarten.

Dies sind jedoch noch nicht alle Änderungen des Zustands der Komponenten, die Einfluss auf die Variable `money` haben. Ein Ladeprotokoll beginnt mit dem Empfang einer `ULoad`-Nachricht durch das Terminal. Das Ladeterminal erhöht den Wert der Variablen `money` und sendet eine `Load`-Nachricht an die Kopierkarte. In einem nachfolgenden Protokollschritt empfängt die Kopierkarte diese `Load`-Nachricht und erhöht ihren Kontostand `balance` entspre-

chend. Da das Erhöhen der Variablen money und balance nicht innerhalb eines Protokollschrittes geschieht, müssen in der Sicherheitseigenschaft zusätzlich die Beträge erfasst werden, die zwar schon zu der Variablen money, aber noch nicht zu der Variablen balance hinzuaddiert wurden. Diese Beträge sind in den Load-Nachrichten gespeichert, die sich in den Inboxen der Kopierkarten befinden, aber noch nicht von diesen verarbeitet wurden. Diese Nachricht führen dazu, dass in einem späteren Protokollschritt, beim Verarbeiten der Nachricht, der Wert der Variablen balance erhöht wird. Weiterhin müssen auch die Load-Nachrichten betrachtet werden, die von dem Angreifer unterdrückt, d.h. aus einer Inbox gelöscht wurden, aber theoretisch zu einem späteren Zeitpunkt wieder eingespielt werden können. An dieser Stelle wird die Tatsache ausgenutzt, dass der Angreifer für die Verbindung zwischen dem Ladeterminal und der Kopierkarte die Fähigkeiten lesen, schreiben und unterdrücken besitzt. Dies bedeutet insbesondere, dass alle Load-Nachrichten, die sich in den Inboxen der Kopierkarten befinden, dem Angreifer bekannt sind, d.h. aus seinem Wissen generiert werden können. Es muss nun die gültige (d.h. von einer Kopierkarte akzeptierte) Load-Nachricht berechnet werden, die den höchsten Wert hat, d.h. bei der der Betrag, der auf die Kopierkarte geladen wird, möglichst hoch ist. Der Angreifer ist in der Lage, alle Load-Nachrichten (mit entsprechendem Betrag amount, Geheimnis passphrase und Nonce challenge) zu erzeugen, wenn der zugehörige Hashwert in seinem Wissen gespeichert ist. Da er das Geheimnis nicht kennt, kann er keine gültigen Hashwerte generieren. Das Prädikat attackerHasLoadHash testet dies.

```
predicates
 attackerHasLoadHash : attackerdataset × Nonce × Secret;

axioms
 attackerHasLoadHash(attacker-known, nonce, secret) ↔
 (∃ i. i ≥ 0.
  hash(mkAuthData(LOAD, secret, nonce, i)) ∈ attacker-known);
```

Während der Protokollausführung kann es passieren, dass es mehr als eine gültige Load-Nachricht für eine Kopierkarte gibt. Verarbeitet die Karte eine gültige Load-Nachricht, ändert sich ihr Zustand und alle weiteren, bis dahin gültigen Load-Nachrichten sind nun nicht mehr gültig. Wenn sich mehrere gültige Load-Nachrichten im Wissen des Angreifers befinden, wird diejenige mit dem höchsten Wert (amount) ausgewählt. Dies reicht aus, da diese Nachricht den Kontostand balance der Kopierkarte maximal erhöht. Die Berechnung der Nachricht mit dem maximalen Betrag ist durch die Funktion getLoadValue mit der Signatur attackerdataset × Nonce × Secret → int beschrieben, das Maximum der leeren Menge ist 0. An dieser Stelle entspricht die Syntax der Formeln nicht der genauen Syntax in KIV. Stattdessen wird eine etwas vereinfachte Syntax verwendet, um die Lesbarkeit zu erhöhen.

```
getLoadValue(attacker-known,nonce,secret) =
max{i : hash(mkAuthData(LOAD,secret,nonce,i)) ∈ attacker-known}
```

Unter Verwendung des zuvor definierten Prädikats und der Funktion kann nun die Sicherheitseigenschaft um alle gültigen Load-Nachrichten (d.h. die von einer (beliebigen) echten

Kopierkarte akzeptiert werden), die dem Angreifer bekannt sind, erweitert werden. Eine Kopierkarte akzeptiert eine Load-Nachricht, wenn sie sich in dem Zustand EXPLOAD befindet und der Hashwert der Nachricht korrekt ist (d.h. die richtige challenge und passphrase verwendet werden). Somit müssen die zu ladenden Beträge (amount) der Load-Nachrichten, die von den Kopierkarten akzeptiert werden, aufsummiert werden. Dies geschieht durch Verwendung einer rekursiven Funktion validLoadMessages mit Signatur (agent → State) × (agent → Nonce) × (agent → Secret) × attackerdataset × nat → int. Listing 9.1 zeigt einen Teil der Axiomatisierung der Funktion.

```
1        attackerHasLoadHash(attacker-known,challenge(cardlet(n)),
2                            passphrase(cardlet(n)))
3     ∧  state(cardlet(n)) = EXPLOAD
4 → validLoadMessages(state,challenge,passphrase,
5                     attacker-known,n)
6    = getLoadValue(attacker-known,challenge(cardlet(n)),
7                   passphrase(cardlet(n)))
8      + validLoadMessages(state,challenge,passphrase,
9                          attacker-known,n-1)
10 ...
```

Listing 9.1: Ausschnitt aus der Axiomatisierung der Funktion validLoadMessages

Im formalen Modell gibt es NUMOFCOPYCARDS konkrete Agenten vom Typ Copycard. Diese werden von eins an durchnummeriert, d.h. es gibt die Karten Copycard(1), Copycard(2), ... , Copycard(n) (siehe Abschnitt 8.2, Seite 198). Die Funktion validLoadMessages bildet die Summe über die Werte amount aller Load-Nachrichten, die von den Kopierkarten (mit den Namen 1 bis NUMOFCARDS) akzeptiert werden. Die Funktion beginnt mit der Berechnung der Werte der Load-Nachrichten für den Agenten Copycard(NUMOFCOPYCARDS) und berechnet anschließend rekursiv die Werte der Load-Nachrichten für die verbleibenden Agenten Copycard(NUMOFCOPYCARDS-1) bis Copycard(1) (Zeilen 8-9). Wenn der Angreifer eine Load-Nachricht kennt, die von der Karte Copycard(n) akzeptiert wird (Zeilen 1-2) und sich diese Kopierkarte im Zustand EXPLOAD befindet (Zeile 3), wird die Summe, die die Werte der gültigen Load-Nachrichten zählt, um den von getLoadValue berechneten Wert erhöht (Zeile 6-7). Dieser Wert ist der höchste Wert, der in einer von Copycard(n) akzeptierten Load-Nachricht vorkommt.

Befindet sich die Kopierkarte nicht im Zustand EXPLOAD oder hat der Angreifer keine gültige Load-Nachricht in seinem Wissen, bleibt die Summe für diese Kopierkarte unverändert und wird rekursiv für alle verbleibenden Karten berechnet.

Auf die gleiche Weise wie beim Aufladen einer Kopierkarte wird der im Hashwert der ResPay-Nachricht gespeicherte Wert amount beim Bezahlen mit einer Karte behandelt. Ähnlich wie im Ladeprotokoll wird beim Bezahlen mit der Karte zunächst im Protokollschritt Pay der Kontostand balance der Karte reduziert und in dem nachfolgenden Protokollschritt ResPay der Wert der Variablen money des Kopiergeräts entsprechend erhöht. Nachdem der Pay- und bevor der ResPay-Schritt ausgeführt wurde, befindet sich die ResPay-Nachricht entweder in der Inbox eines Kopiergeräts (und wartet auf Verarbeitung) oder wurde vom Angreifer unterdrückt. Wie bei den Load-Nachrichten müssen die Werte der ResPay-Nachrichten ebenfalls

in die Sicherheitseigenschaft mit aufgenommen werden. Deshalb wird analog zu der Funktion `validLoadMessages` eine Funktion `validResPayMessages` definiert, die alle `ResPay`-Nachrichten betrachtet, die aktuell von einem Kopiergerät akzeptiert werden. Die Funktion bildet die Summe über alle Werte (für alle Kopiergeräte), die zu der Variablen `money` addiert werden, wenn der entsprechende Schritt `ResPay` ausgeführt wird.

Zuletzt müssen noch alle Beträge summiert und in die Sicherheitseigenschaft mit aufgenommen werden, die beim Aufladen einer Kopierkarte zwar bereits zu dem Wert der Variablen `money` addiert wurden, aber noch nicht in den `validLoadMessages` gespeichert sind, weil vor dem eigentlichen Ladevorgang noch die Authentisierung zwischen Karte und Kopiergerät durchgeführt werden muss (Schritte `Authenticate` und `ResAuthenticate`). Ähnlich wie bei der Berechnung der Funktion `validLoadMessages` werden alle Agenten eines Typs, in diesem Fall des Typs `DepositMachine`, betrachtet. Es wird nun die Summe über den Werten der Variablen `value` aller Ladeterminals berechnet, die sich aktuell im Zustand EXPRESAUTH befinden.

```
  state(terminal(n)) = EXPRESAUTH
→
  allValues(...,n) = value(DepositMachine(n)) + allValues(...,n-1)
```

Wenn sich das Ladeterminal `DepositMachine(n)` nicht im Zustand EXPRESAUTH befindet, wird der Wert dieser Terminals ignoriert und rekursiv die anderen Ladeterminals betrachtet.

Die veränderte Sicherheitseigenschaft ist dann wie folgt spezifiziert:

```
   allIssued(issued)
≥ allCollected(collected)
   + allBalance(balance)
   + validLoadMessages(..,attacker-known,NUMOFCOPYCARDS)
   + validResPayMessages(..,value,attacker-known,
                                      NUMOFCOPYINGMACHINES)
   + allValues(value,state,NUMOFDEPOSITMACHINES)
```

Die Ungleichung beachtet alle Änderungen von Variablen, die für das Erhöhen und Reduzieren der Werte der Variablen `money` (in einem späteren Protokollschritt) relevant sind. Wird zum Beispiel der Kontostand `balance` einer Karte beim Bezahlen reduziert, werden im selben Schritt die `validResPayMessages` um den gleichen Betrag erhöht. Wird die ResPay-Nachricht verarbeitet und der Protokollschritt `ResPay` ausgeführt, werden die `validResPayMessages` um den in der verarbeiteten Nachricht enthaltenen Wert reduziert (weil das Kopiergerät die Nachricht nun aufgrund des geänderten Zustands nicht mehr akzeptiert). Dafür wird der Wert der Variablen `money` des Kopiergeräts in diesem Schritt und den Wert erhöht. Die veränderte Sicherheitseigenschaft notiert sozusagen, wie die zu ladenden bzw. abzubuchenden Beträge während der Ausführung der Protokolle durch die Komponenten „wandern“.

Jetzt ist es möglich die veränderte Sicherheitseigenschaft als invariant bezüglich einem STEP nachzuweisen. Hierfür wird symbolische Ausführung verwendet, die Standardbeweisstrategie für Dynamische Logik in KIV. Die zu verifizierende Eigenschaft lautet:

```
    Base-INV(..)
  ∧ allIssued(issued)
    ≥ allCollected(collected)
       + allBalance(balance)
       + validLoadMessages(..,attacker-known,NUMOFCOPYCARDS)
       + validResPayMessages(..,value,attacker-known,
                                          NUMOFCOPYINGMACHINES)
       + allValues(value,state,NUMOFDEPOSITMACHINES)
⊢ [STEP]
  allIssued(issued)
  ≥ allCollected(collected)
     + allBalance(balance)
     + validLoadMessages(..,attacker-known,NUMOFCOPYCARDS)
     + validResPayMessages(..,value,attacker-known,
                                        NUMOFCOPYINGMACHINES)
     + allValues(value,state,NUMOFDEPOSITMACHINES)
```

Hat man diese Invariante bewiesen, lässt sich die ursprüngliche Eigenschaft `allMoney(money)` ≥ `0` ebenfalls beweisen.

10 Verfeinerung zwischen Code und formalem Modell

Zusammenfassung: Die auf dem formalen Modell bewiesenen Sicherheitseigenschaften gelten nicht automatisch auch für den generierten Code. Eine Voraussetzung hierfür ist, dass der generierte Code eine Verfeinerung des generierten formalen Modells ist. Beim Entwurf der Modellierungsrichtlinien sowie der Implementierung der Modelltransformationen wurde ein besonderes Augenmerk darauf gerichtet, dass diese Verfeinerungsbeziehung gilt. In diesem Kapitel wird begründet, weshalb der generierte Quellcode eine Verfeinerung der formalen Spezifikation ist und warum die auf formaler Ebene bewiesenen Eigenschaften sich auf den Quellcode übertragen. Eine formale Verifikation der Modelltransformationen, die diese Verfeinerungsbeziehung garantiert, ist nicht Teil dieser Arbeit.

Im Folgenden wird zunächst in Abschnitt 10.1 der verwendete Verfeinerungsbegriff erläutert. Abschnitt 10.2 gibt einen Überblick über die unterschiedlichen Aspekte, für die die Äquivalenz- bzw. Verfeinerungsbeziehung betrachtet werden muss. Dabei wird erläutert, für welche dieser Punkte die Korrespondenz zwischen Code und formalen Modell offensichtlich ist und für welche eine besondere Behandlung erforderlich war, damit die Beziehung gilt. Auf deren Umsetzung wird anschließend in Abschnitt 10.3 genauer eingegangen. Abschnitt 10.4 setzt die in diesem Kapitel beschriebenen Ergebnisse in den Kontext anderer Forschungsarbeiten zu diesem Thema.

Wie auch in Kapitel 6 wird an den Stellen, an denen sich der generierte Java-Code nicht von dem generierten Java Card-Code unterscheidet, von Java-Code gesprochen. Gibt es einen Unterschied, werden die Begriffe Terminal-Code und Smart Card-Code verwendet.

10.1. Verfeinerung

Bei der Verifikation von Eigenschaften auf einem abstrakten Modell stellt sich die Frage, ob die bewiesenen Eigenschaften auch für eine Implementierung, die das abstrakte Modell realisiert, gelten. Dies ist nur dann der Fall, wenn die Implementierung korrekt bezüglich der abstrakten Spezifikation ist und keine zusätzlichen Sicherheitslücken durch die Implementierung entstehen. In der Dissertation von Grandy [64] wurde eine allgemeine Verfeinerungs-

theorie entwickelt, mit der die Korrektheit der Implementierung sowie die Übertragung der auf abstrakter Ebene bewiesenen Eigenschaften auf den Code sichergestellt werden kann. Die Methodik wurde anhand verschiedener Fallstudien mit manuell geschriebenem Code evaluiert und ist verwendbar, wenn ein konkretes formales Modell sowie der Java-Code einer Anwendung gegeben sind. Dann ist der Nachweis möglich, dass diese Implementierung eine Verfeinerung der entsprechenden abstrakten Spezifikation ist. In dieser Dissertation soll jedoch eine Aussage darüber gemacht werden, dass die Korrektheit, d.h. die Verfeinerungsbeziehung, für jede mit dem SecureMDD-Ansatz entwickelte Anwendung gilt. Der von Grandy in [64, 66, 68] verwendete Verfeinerungsbegriff ist dabei auch auf den SecureMDD-Ansatz anwendbar und wird im Folgenden kurz erläutert.

Abbildung 10.1 zeigt die in [64,66] definierte und verwendete Verfeinerungsbeziehung zwischen dem formalen Modell und dem Code.

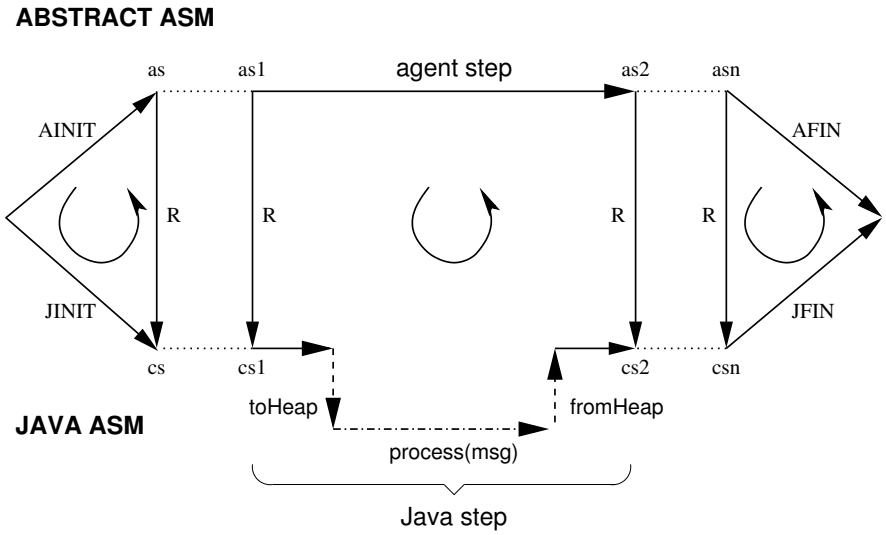

Abbildung 10.1.: Verfeinerung zwischen dem abstrakten Modell und dem Java-Code (aus [66], S. 226)

Für den Code wird dabei eine Java-ASM konstruiert. In dieser werden die ASM-Regeln der abstrakten ASM schrittweise durch den entsprechenden Java-Code ersetzt. Dies bedeutet, dass in jedem Schritt eine Komponente sowie die abstrakte Spezifikation ihrer Protokollschritte durch den Code, d.h. die entsprechende `process`-Methode, ersetzt werden. Die in diesem Schritt nicht verfeinerten Komponenten bleiben dabei unverändert. Die Java-ASM ist somit eine Mischung aus abstrakter Spezifikation und Java-Code. Die Java-ASM ist eine 1-1 Verfeinerung der abstrakten ASM. Da diese ASM den Java-Code einiger Komponenten beinhaltet, muss zusätzlich eine Übersetzung der abstrakten Datentypen in Java-Objektstrukturen und umgekehrt angegeben werden. Diese ist durch die ASM-Regeln `toHeap` für die Abbildung abstrakter Datentypen in Java-Objektstrukturen und `fromHeap` für die Rücktransformation definiert.

Für die eigentlichen Protokollschritte der mit SecureMDD entwickelten Anwendungen gilt, dass der generierte Quellcode semantisch äquivalent zu dem formalen Modell ist. Die Äquivalenzbeziehung bezieht sich auf verschiedene statische sowie dynamische Aspekte. An vielen

Stellen sind die Übereinstimmungen klar sichtbar. Beispiele hierfür sind die Abbildung des Kontrollflusses (eine Fallunterscheidung z.B. ist sowohl im Code als auch im formalen Modell durch ein if-then-else abgebildet) oder die Abbildung der eigentlichen Komponentenklassen, die im Code in Java-Klassen und im formalen Modell in abstrakte Datentypen übersetzt werden. Für andere Aspekte ist das Herstellen einer semantischen Äquivalenz jedoch eine Herausforderung und hat beim Entwurf der Modellierungsrichtlinien und bei der Erstellung der Modelltransformationen ein besonders sorgfältiges Vorgehen erfordert. Dies sind zum Beispiel die Umsetzung der Kopiersemantik auf Codeebene oder die Objektverwaltung im generierten Code. Für die Kommunikationsstruktur gilt, dass der generierte Code eine Verfeinerung des formalen Modells ist. Das Ergebnis des Sicherstellens der semantischen Äquivalenz bzw. Verfeinerungsbeziehung ist die automatische Generierung von korrektem Quellcode, der die auf formaler Ebene bewiesenen Sicherheitseigenschaften erfüllt.

10.2. Überblick über die zu beachtenden Aspekte

Die folgende Tabelle listet alle Aspekte der Generierung auf, die bei der Frage nach der Äquivalenz bzw. Verfeinerung zwischen dem Code und dem formalen Modell betrachtet werden müssen.

	UML und MEL	Terminal-Code	Smart Card-Code	formale Spezifikation	
1.	primitiver Datentyp	Java-Typ	Java Card-Typ	abstrakter Datentyp	X
2.	Daten- o. Nachrichtenklasse	Java-Klasse	Java-Klasse	abstrakter Datentyp	X
3.	Komponentenklasse	Java-Klasse	Java-Klasse (extends Applet)	abstrakter Datentyp (Agent)	✓
4.	Attribute und Assoziationen der Komponenten	Felder	Felder	dynamische Funktionen	✓
5.	Protokollschritt im Aktivitätsdiagramm	Java-Methode	Java-Methode	ASM-Regel (Prozedur)	✓
6.	Vordefinierte MEL-Operationen	Java-Methoden	Java-Methoden	algebraische Funktionen und Prozeduren	X
7.	Kontrollfluss	Kontrollfluss	Kontrollfluss	Kontrollfluss	✓
8.	Kopiersemantik MEL	Referenzsemantik	Referenzsemantik	Kopiersemantik	X
9.	Erzeugen eines neuen Objekts mit `create`	Erzeugen eines neuen Objekts mit `new`	Herausgabe eines Objekts durch den Objektmanager	Erzeugen eines Elements des entsprechenden Datentyps	X
10.	Lokale Variablen in MEL	Lokale Variablen	Lokale Variablen	Lokale Variablen	✓

	UML und MEL	Terminal-Code	Smart Card-Code	formale Spezifikation	
11.	Zugriff auf Attribute, Assoziationen und Variablen in MEL	Zugriff auf Felder und Variablen in Java	Zugriff auf Felder und Variablen in Java	Lookup in dynamischen Funktionen, Anwendung von Selektoren und Zugriff auf lokale Variablen	✓
12.	Aufruf eines Subdiagramms	Aufruf einer Methode	Aufruf einer Methode	Aufruf einer Prozedur	✓
13.	Kommunikation zwischen Terminal und Smart Card	Versand bzw. Empfang eines Byte-Arrays	Versand bzw. Empfang eines Byte-Arrays	Schreiben einer Nachricht in die Inbox	X
14.	Abbruch eines Protokollschritts (durch `FlowFinal` oder implizit durch eine MEL-Operation)	Abbruch des Protokollschritts durch Werfen einer Exception	Abbruch des Protokollschritts durch Werfen einer Exception und Senden eines Fehlercodes an das Terminal	Abbruch des Protokollschritts durch Beenden der Ausführung der ASM-Regel	X
15.	kein Auftreten von Null	Null	Null	kein Auftreten von Null	X
16.	keine Laufzeitfehler	Laufzeitfehler	Laufzeitfehler	keine Laufzeitfehler	X
17.	Angabe der Angreiferfähigkeiten	–	–	Angreifer	✓

Die zweite Spalte der Tabelle bezieht sich auf die plattformunabhängige Modellierung mit UML und MEL. In der dritten Spalte ist die Umsetzung des jeweiligen Aspekts im Terminal-Code und in der vierten Spalte die Umsetzung im Smart Card-Code aufgelistet. Die fünfte Spalte enthält die Umsetzung im formalen Modell und die sechste Spalte gibt an, ob die Äquivalenzbeziehung offensichtlich gilt (✓) oder ob zusätzliche Arbeit nötig war, um diese Beziehung herzustellen (X). Diese Aspekte sowie ihre Umsetzung werden in Abschnitt 10.3 im Detail erläutert.

Die Tabelle wird nachfolgend erläutert:

1. Die in UML verwendeten primitiven Datentypen `Number`, `Boolean` und `String` werden im Code auf entsprechende Java- bzw. Java Card-Typen abgebildet. Gesondert betrachtet werden muss der Typ `Number`, der im Terminal-Code sowie im formalen Modell auf Integer und im Smart Card-Code auf den primitiven Typ `short` abgebildet wird. Außerdem erfordert der Typ `String` eine besondere Behandlung. Dieser wird im Smart Card-Code auf Byte-Arrays abgebildet, während im Terminal-Code und im formalen Modell Strings verwendet werden.

2. Eine Daten- oder Nachrichtenklasse im UML-Modell wird im Code auf eine Java-Klasse und im formalen Modell auf einen abstrakten Datentyp abgebildet. Damit eine Java-Klasse auf einen abstrakten Datentyp semantisch äquivalent abbildbar ist, müssen zusätzliche Einschränkungen für die Daten- und Nachrichtenklassen sowie ihre Attribute und Assoziationen gelten. Diese sind die Zyklenfreiheit der Klassen, die Nullfreiheit der Instanzen zur Laufzeit sowie die Einhaltung der Sharing-Freiheit der Objekte.
3. Eine Komponentenklasse wird sowohl im Terminal- als auch im Smart Card-Code auf eine Java-Klasse abgebildet. Im Smart Card-Code ist diese Klasse von der Klasse `Applet` des Java Card-Frameworks abgeleitet. Komponentenklassen werden im formalen Modell durch den abstrakten Datentyp `agent` repräsentiert. Dies ist ein Summentyp über alle Komponentenklassen. Für die Äquivalenzbeziehung muss hier nichts weiter beachtet werden.
4. Jede Komponentenklasse besitzt einen Zustand, der im UML-Modell durch die Attribute und Assoziationen der Klasse beschrieben ist. Dieser wird im Java-Code auf Felder der jeweiligen Klasse abgebildet. In der formalen Spezifikation wird der Zustand durch dynamische Funktionen angegeben, d.h. es wird eine dynamische Funktion für jedes Attribut und Assoziationsende eines Agenten erzeugt. Die Äquivalenz ist an dieser Stelle trivial.
5. Ein Protokollschritt ist im UML-Modell in einem Aktivitätsdiagramm modelliert. Im Java-Code gibt es für jeden Protokollschritt eine eigene Methode, die die Implementierung dieses Schrittes enthält. Im formalen Modell wird für jeden Protokollschritt eine ASM-Regel generiert, die den Schritt durchführt. Technisch ist eine ASM-Regel eine Prozedur. Für die Äquivalenzbeziehung muss hier nichts weiter beachtet werden.
6. In MEL gibt es eine Reihe vordefinierter kryptographischer Operationen sowie Operationen für den Zugriff auf Listen. Diese Operationen werden im Java-Code auf Methoden abgebildet. Im formalen Modell sind diese durch algebraisch spezifizierte Funktionen sowie Prozeduren realisiert. Bezüglich der Äquivalenzbeziehung sind einige Punkte zu beachten. Die kryptographischen Operationen werden im Code auf Methoden der Java bzw. Java Card Crypto API abgebildet. Diese operieren auf Byte-Arrays, während die entsprechenden Funktionen und Prozeduren im formalen Modell direkt auf den Elementen der entsprechenden abstrakten Datentypen aufgerufen werden. Weiterhin muss für das formale Modell die perfekte Kryptographie vorausgesetzt werden. Die Frage ist, welche Annahmen an dieser Stelle für die Implementierung gemacht werden können. Eine Liste wird im generierten Code durch ein Byte-Array repräsentiert, während im formalen Modell eine entsprechende Listenspezifikation generiert wird. Für die Äquivalenzbeziehung muss gezeigt werden, dass beide Umsetzungen äquivalent sind.
7. Die Modellierung mit UML und MEL erlaubt bedingte Fallunterscheidungen sowie die Hintereinanderausführung von Ausdrücken. Diese Kontrollfluss-Elemente werden durch if-then-else-Statements bzw. Hintereinanderausführung im Java-Code und im formalen Modell unterstützt. Bezüglich der Äquivalenz ist hier nichts zu beachten.
8. MEL hat eine Kopiersemantik. Dies bedeutet, dass ein Objekt bei Zuweisung an ein Attribut, ein Assoziationsende oder eine lokale Variable kopiert und diese Kopie gespeichert wird. Java und Java Card dagegen besitzen eine Referenzsemantik, d.h. bei Zuweisungen und Aktualisierungen einzelner Felder eines Objekts werden Referenzen

auf das zugewiesene Objekt gespeichert. Die gespeicherten Objekte können somit per Seiteneffekt nachträglich noch verändert werden. Das formale Modell besitzt, wie MEL, eine Kopiersemantik. An dieser Stelle ist deshalb eine Kopiersemantik auch auf Codeebene zu erzwingen, damit die Äquivalenzbeziehung gilt.

9. Das Erzeugen eines neuen Objekts erfolgt in der Modellierung mit MEL durch Verwendung des Schlüsselwortes `create`. Im Terminal-Code werden Objekte äquivalent dazu mit dem Java-Schlüsselwort `new` erzeugt. Im formalen Modell werden entsprechend Elemente der abstrakten Datentypen erzeugt. Besondere Beachtung benötigt an dieser Stelle der Smart Card-Code. Aufgrund der fehlenden Garbage Collection auf einer Smart Card sowie dem begrenzten Speicherplatz, ist es nicht möglich, zur Laufzeit beliebig viele neue Objekte zu erzeugen. Dies ist in SecureMDD durch die Verwendung eines Objektmanagers gelöst. Die Äquivalenzbeziehung muss an dieser Stelle näher betrachtet werden.
10. In MEL ist es möglich, innerhalb eines Protokollschritts eine lokale Variable zu deklarieren, die bis zum Ende des Protokollschritts gültig ist. Der Java-Code deklariert an den entsprechenden Stellen ebenfalls lokale Variablen. Diese sind bis zum Ende der Methode, in der sie deklariert wurden, d.h. bis zum Ende des Protokollschritts, gültig. Im formalen Modell wird eine lokale Variable durch ein LET-Konstrukt deklariert, für das explizit ein Body als Bindungsbereich angegeben werden muss. Der Body wird bei der Generierung so gewählt, dass die lokale Variable ebenfalls bis zum Ende des Protokollschritts, d.h. bis zum Ende der ASM-Regel, gültig ist. Darüber hinaus ist bezüglich der Äquivalenz nichts zu beachten.
11. In MEL ist der Zugriff auf Attribute und Assoziationen sowie auf lokale Variablen möglich. Im Java-Code entspricht dies dem Zugriff auf Felder und Variablen. Im formalen Modell entspricht der Zugriff auf Attribute und Assoziationen einer Komponentenklasse dem Lookup in der entsprechenden dynamischen Funktion. Der Zugriff auf die Attribute und Assoziationen einer Daten- oder Nachrichtenklasse entspricht der Anwendung eines Selektors. Um an dieser Stelle die gleiche Syntax wie in MEL und Java zu ermöglichen, werden entsprechende algebraische Funktionen definiert. Die Äquivalenzbeziehung ist jedoch trivial.
12. In MEL ist der Aufruf eines Subdiagramms aus einem Protokollschritt heraus möglich. Dieser Aufruf wird im Java-Code in den Aufruf der Methode, die das Subdiagramm implementiert, übersetzt. Im formalen Modell wird an dieser Stelle die Prozedur aufgerufen, die das Subdiagramm realisiert. Bezüglich der Äquivalenzbeziehung ist nichts weiter zu beachten.
13. Die Kommunikation zwischen einem Terminal und einer Smart Card ist in UML durch den Austausch von Nachrichten modelliert. Im Java-Code ist das Versenden von Nachrichtenobjekten nicht möglich, da die Kommunikation mit einer Smart Card auf Byte-Arrays basiert. Dies bedeutet, dass es im Code zusätzlich eine (De-)Serialisierungsschicht gibt, die im formalen Modell nicht vorhanden ist. Dort entspricht das Senden dem Schreiben des Nachrichtenelements in die entsprechende Inbox des Empfängers, der Empfang entspricht der Entnahme der Nachricht aus der Inbox. Zusätzlich ist die Kommunikation in der generierten Implementierung synchron realisiert, während das formale Modell ein asynchrones Kommunikationsmodell verwendet. Für die Verfeinerungsbeziehung müssen diese Aspekte betrachtet werden.

14. In der Modellierung kann der Abbruch eines Protokollschritts explizit oder implizit stattfinden. Explizit geschieht dies durch Modellierung eines `FlowFinal` Knotens. Implizit bricht ein Protokollschritt durch Auftreten eines Fehlers bei der Durchführung einer vordefinierten MEL-Operation ab. Im Terminal-Code wird ein Protokollschritt durch eine (SecureMDD-)Exception abgebrochen. Im Smart Card-Code wird ein Protokollschritt durch das Werfen einer (ISO7816.SW_CONDITIONS_NOT_SATISFIED-) Exception und dem Senden eines Fehlercodes an das entsprechende Terminal abgebrochen. Im formalen Modell wird der Protokollschritt ebenfalls abgebrochen, d.h. der ASM-Schritt beendet.
15. In MEL sind alle Felder und Variablen ungleich `Null`, d.h. sie verweisen auf ein Objekt, und es gibt kein `Null`-Literal. Dies gilt ebenfalls für das formale Modell. Auch hier werden alle dynamischen Funktionen und Variablen mit Werten ungleich `Null` initialisiert. Im Java-Code können Objekte prinzipiell `Null` sein, d.h. auf kein Objekt verweisen. Damit die Äquivalenzbeziehung gilt, muss sich der Code genauso verhalten wie das formale Modell, d.h. ebenfalls `Null`-frei sein.
16. In MEL können keine Laufzeitfehler, wie z.B. das Überschreiten von Arraygrenzen, auftreten. Auch im formalen Modell gibt es keine Laufzeitfehler. Im Java-Code können jedoch prinzipiell Laufzeitfehler auftreten. Für die Äquivalenz muss sichergestellt sein, dass der generierte Code ebenfalls keine Laufzeitfehler produziert.
17. Der Angreifer bzw. dessen Fähigkeiten sind im UML-Deploymentdiagramm durch den Stereotyp ≪Threat≫ modelliert. Im formalen Modell ist der Angreifer als eigener Agent modelliert, der entsprechend seines Wissens und seiner Fähigkeiten agieren kann. Für den Angreifer wird natürlich kein Quellcode generiert. Die Nichtbetrachtung des Angreifers im Code hat jedoch keinen Einfluss auf die Äquivalenzbeziehung.

10.3. Besondere Herausforderungen und deren Lösung

In diesem Abschnitt werden die Aspekte näher betrachtet, für die die Äquivalenzbeziehung (bzw. Verfeinerung für die Kommunikationsstruktur) nicht offensichtlich gilt.

10.3.1. Primitive Datentypen

Der im UML-Modell verwendete, unbeschränkte Typ `Number` wird bei der Generierung des Terminal-Codes auf den primitiven Java-Typ `int` abgebildet. In der formalen Spezifikation wird ebenfalls der Typ `int` verwendet. Da Java Card keine Integer unterstützt, wird im generierten Smart Card-Code der Typ `short` verwendet. `short`-Werte haben einen kleineren Wertebereich als Integer. Damit sich das formale Modell und der Smart Card-Code dennoch gleich verhalten, sind zwei Dinge zu beachten:

- Beim Aufruf von arithmetischen Operationen kann das Ergebnis außerhalb des Wertebereichs eines `shorts` liegen. Deshalb wird im Smart Card-Code für jede arithmetische Operation eine Methode generiert, die überprüft, ob es bei Durchführung der Operation zu einem Over- bzw. Underflow kommt. Ist dies der Fall, wirft die Smart Card eine Exception, sendet einen Fehlercode an das Terminal und bricht den Protokollschritt ab, ohne die Operation auszuführen. Die Abstract State Machine verhält sich bei der Aus-

führung arithmetischer Operationen auf einer Smart Card-Komponente so, als würde sie mit `short`-Werten rechnen. Dies ist folgendermaßen gelöst: In der formalen Spezifikation wird der Test auf Auftreten eines Over- bzw. Underflows vor Ausführen der Operation durchgeführt und im Falle eines positiven Tests wird die Ausführung der ASM-Regel und somit der gesamte Schritt beendet. Dies ist äquivalent zum Abbrechen des Protokollschritts im generierten Smart Card-Code. So wird beispielweise der MEL-Ausdruck `balance := balance + value` in dem UML-Modell der Kopierkartenanwendung im generierten Smart Card-Code ersetzt durch den Java-Ausdruck `balance = Math.plus(balance, value);`. Die Methode `plus` prüft zunächst, ob bei Durchführung der Operation ein Over- oder Underflow auftritt und wirft dann ggf. eine SecureMDD-Exception. Tritt kein Over- oder Underflow auf, wird die Addition durchgeführt und die `plus`-Methode gibt das Ergebnis zurück. Die Übersetzung in das formale Modell, d.h. einen Teil einer ASM-Regel, geschieht an dieser Stelle folgendermaßen:

```
..
if shortOverflow(balance(ag) + value-var)
then STOPSTEP(; exception_occurred)
else { balance(ag) := balance(ag) + value-var ;}
..
```

Das Prädikat `shortOverflow` prüft für einen Integer (der das Ergebnis einer arithmetischen Operation ist), ob der Wert außerhalb des gültigen Wertebereichs eines `short`-Wertes liegt und somit ein Over- bzw. Underflow auftreten würde. Tritt ein Over- oder Underflow auf, wird die ASM-Regel `STOPSTEP` aufgerufen (siehe 10.3.7), ansonsten wird die Operation ausgeführt. Somit gilt, dass an den Stellen, an denen der Code eine Exception wirft und somit den Protokollschritt beendet, die ASM-Regel nicht weiter ausgeführt wird. Aus diesem Grund gilt, dass der generierte Code und das formale Modell an dieser Stelle äquivalent sind. Für die Addition wurde dies beispielhaft mithilfe des Java-Kalküls in KIV bewiesen.

- Das Terminal kann einen Integer in einer Nachricht an eine Smart Card senden. Dort muss dieser bei der Deserialisierung des zugehörigen Byte-Arrays als `short`-Wert abgespeichert werden. Liegt der empfangene Wert außerhalb des Wertebereichs eines `shorts`, wirft die Smart Card eine Exception, sendet einen Fehlercode zurück an das Terminal und bricht die Ausführung des Protokollschritts ab. Die ASM muss sich an dieser Stelle genauso verhalten wie der Code, den Test auf Over- bzw. Underflow ebenfalls durchführen und ggf. den Protokollschritt abbrechen (siehe Abschnitt 10.3.6).

Der im UML-Modell verwendete primitive Typ `String` wird im Smart Card-Code durch ein Byte Array repräsentiert. Das Byte Array enthält dabei die ASCII-Kodierung des Strings. Aus diesem Grund müssen die im SecureMDD-Ansatz verwendeten Strings aus Zeichen bestehen, für die es eine ASCII-Repräsentation gibt. Ein Zeichen wird immer durch ein Byte repräsentiert. Bei Verwendung der ASCII-Kodierung ist die Abbildung von Strings auf Byte-Arrays injektiv. Da jeder String eine eindeutige ASCII-Repräsentation besitzt, ist die Verwendung des KIV Basistyps `string` im formalen Modell möglich, auch wenn der entsprechende Smart Card-Code Byte-Arrays verwendet. MEL definiert keine Methoden für die Manipulation von Strings (wie zum Beispiel Konkatenation). Deshalb gibt es auch im generierten Code sowie dem formalen Modell keine Operationen für die Manipulation von Strings, deren Verwendung

einen Einfluss auf die Verfeinerungsbeziehung zwischen formaler Spezifikation und Code haben könnte.

10.3.2. Daten- und Nachrichtenklassen

Die im UML-Klassendiagramm definierten Daten- und Nachrichtenklassen werden im Code auf Java-Klassen und im formalen Modell auf algebraische Datentypen abgebildet. Um Java-Klassen durch abstrakte Datenstrukturen repräsentieren zu können, müssen in der Modellierung und bei der Generierung des Codes einige Punkte berücksichtigt werden:

- Die in UML modellierten Daten- und Nachrichtenklassen dürfen keine Zyklen enthalten. Ein Klassendiagramm ist zyklenfrei, wenn es nicht möglich ist, von einer beliebigen Klasse ausgehend den gerichteten Assoziationen (über mehrere Klassen hinweg) zu folgen und die Ausgangsklasse wieder zu erreichen. Die Einhaltung der Zyklenfreiheit wird während der Validierung des UML-Modells überprüft, die vor der eigentlichen Generierung stattfindet. Die entsprechenden Klassen im generierten Code sind dann ebenfalls zyklenfrei.
- Die Codegenerierung garantiert, dass der gesamte Code sharingfrei ist. Das heißt, es kommt nicht vor, dass es zwei Referenzen auf dasselbe Objekt gibt. Dies ist so implementiert, dass bei Zuweisungen die zugewiesenen Objekte kopiert werden, anstatt wie in Java üblich, eine Referenz auf das Objekt zu speichern (siehe Abschnitt 10.3.4).
- Der generierte Code muss Null-frei sein. Dies bedeutet, dass alle Felder und Variablen zur Laufzeit ungleich `Null` sind und dass das `Null`-Literal auch im Code nicht vorkommt. Dies impliziert zum Beispiel, dass die Java-Methoden niemals `Null` zurückgeben dürfen. Die `Null`-Freiheit des Codes ist durch die `Null`-Freiheit der Sprache MEL und eine entsprechende Übersetzung von MEL in Code sichergestellt (siehe Abschnitt 10.3.8).

Die Einhaltung dieser drei Punkte bewirkt, dass die Objektstruktur bei Ausführung des generierten Codes kein Graph, sondern ein Baum ist und alle Felder sowie lokale Variablen initialisiert, d.h. nicht `Null`, sind. So ist es möglich, die in der Implementierung verwendeten Datentypen auf algebraische Datentypen abzubilden.

10.3.3. Vordefinierte Operationen

Kryptographische Operationen

Für die in MEL definierten kryptographischen Operationen, z.B. zum Verschlüsseln und Signieren eines Datums, werden automatisch entsprechende Java-Methoden sowie im formalen Modell algebraische Funktionen sowie Prozeduren generiert. Im formalen Modell wird dabei die Annahme der *perfekten Kryptographie* getroffen (siehe Abschnitt 8.2, Seite 187).

Die Implementierung der kryptographischen Operationen verwendet die kryptographischen Methoden, die von der Java Crypto API bzw. von der Java Card Crypto API zur Verfügung gestellt werden. Für die Verfeinerung muss die Frage beantwortet werden, wie die im formalen Modell verwendete Annahme der perfekten Kryptographie auf Implementierungsebene gehandhabt wird. In der Praxis sind die verwendeten Verschlüsselungs- und Hashfunktionen

nicht injektiv, d.h. es lassen sich z.B. zwei Klartexte finden, die denselben Hashwert haben. Diese Frage wurde bereits in der Dissertation von Grandy [64] diskutiert und beantwortet. Dort wird die perfekte Kryptographie auch für die Implementierung angenommen. Dies ist damit begründet, dass die Wahrscheinlichkeit für das Auftreten von konkreten Eingaben, die die perfekte Kryptographie verletzen (zum Beispiel das Auftreten von Kollisionen bei Hashfunktionen), sehr gering ist. In dieser Arbeit werden außerdem nur logische Eigenschaften von Protokollen betrachtet und keine quantitativen Aspekte (wie z.B. die Suchraumgröße), die andere Techniken erfordern. Die von Grandy in [64] gemachten Annahmen gelten auch für den SecureMDD-Ansatz und die perfekte Kryptographie kann somit ebenfalls auch für die Implementierung angenommen werden.

Die Annahme der perfekten Kryptographie in der Implementierung ist beispielhaft anhand der symmetrischen Verschlüsselung eines Datums illustriert. Ein symmetrisch verschlüsseltes Datum kann genau dann entschlüsselt werden, wenn der Schlüssel, mit dem entschlüsselt wird, gleich dem Schlüssel ist, der für die Verschlüsselung verwendet wurde. Dies ist im formalen Modell in dem Prädikat `canDecrypt` definiert:

```
   canDecrypt(symmkey1, mkEncDataSymm(symmkey2, plaindata))
↔ symmkey1 = symmkey2;
```

Außerdem muss der Typ des entschlüsselten Datums dem erwarteten Typ entsprechen, damit keine Java `ClassCastException` auftreten kann (siehe Abschnitt 10.3.9). Die `decrypt`-Operation darf deshalb nur als Initialausdruck für eine lokale Variable verwendet werden und das entschlüsselte Datum muss denselben Typ wie die Variable haben

Bevor in der ASM die `decrypt`-Operation aufgerufen wird, wird deshalb geprüft, ob das Prädikat `canDecrypt` und der Typcheck wahr sind. In diesem Fall wird die `decrypt`-Operation ausgeführt, anderenfalls die Prozedur `STOPSTEP` aufgerufen und der Protokollschritt wird nicht weiter ausgeführt.

Im generierten Code ist es natürlich so, dass der Schlüssel, mit dem verschlüsselt wurde, der Komponente, die das Datum entschlüsselt, nicht bekannt ist. In der Implementierung basiert die Verschlüsselung auf Byte-Arrays. Das entschlüsselte Datum liegt in Form eines Byte-Arrays vor. Die Annahme ist, dass ein entschlüsseltes Byte-Array, das fehlerfrei deserialisiert werden kann und außerdem dem erwarteten Typ entspricht, mit dem richtigen Schlüssel entschlüsselt wurde. Diese Annahme beruht auf der Tatsache, dass ein mit einem falschen Schlüssel entschlüsseltes Datum mit sehr hoher Wahrscheinlichkeit keine gültige Serialisierung des erwarteten Klartextobjekts ist.

Gesondert zu betrachten sind außerdem die Operationen für das Generieren einer neuen Nonce sowie das Generieren eines kryptographischen Schlüssels. Eine Nonce ist eine Zufallszahl, die nur einmal, d.h. in einem Kontext, verwendet werden sollte. Für die Sicherheit der Protokolle ist es essentiell, dass eine Nonce nicht vorhersehbar ist und zur Lebenszeit der Anwendung niemals dieselbe Nonce ein zweites Mal generiert wird. Die formale Spezifikation der Methode `generateNonce` berücksichtigt diese beiden Punkte. In der Implementierung ist hierfür die Verwendung eines kryptographisch sicheren Zufallszahlengenerators notwendig. Für das Generieren von kryptographischen Schlüsseln werden ebenfalls Zufallszahlen verwendet. Im formalen Modell werden hierfür Elemente aus dem Pool von Nonces genommen (und dann aus diesem gelöscht), der auch für das Generieren von Nonces verwendet wird. In der Implementierung sind jedoch nicht alle generierbaren Nonces für die Verwendung als

Schlüssel geeignet, z.B. bei der Verwendung von RSA für asymmetrische Schlüsselpaare. Die Implementierung beachtet dies allerdings und liefert bei jedem Aufruf der `generateKey()`- bzw. `generateKeyPair()`-Methode einen gültigen Schlüssel bzw. ein gültiges Schlüsselpaar. Aus diesem Grund kann im formalen Modell angenommen werden, dass alle erzeugten Schlüssel(paare) gültig sind.

Ein weiterer Unterschied zwischen dem formalen Modell und dem Code ist, dass die kryptographischen Operationen in der Implementierung auf Byte-Arrays operieren. Folgende Grafik 10.2 zeigt beispielhaft die Aufrufreihenfolge der Methoden, die für das Verschlüsseln eines Objekts (dass das Interface `PlainData` implementiert) benötigt werden. Alle anderen Operationen sind analog implementiert.

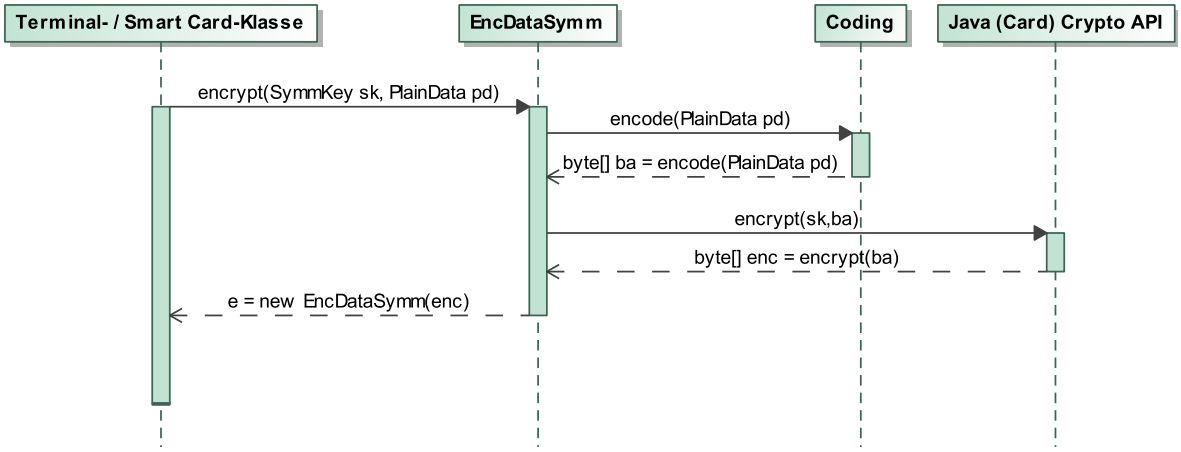

Abbildung 10.2.: Aufrufreihenfolge der Methoden beim Verschlüsseln eines Objekts

Eine Terminal- oder Smart Card-Klasse möchte ein Objekt verschlüsseln. Dies wird in einem UML-Aktivitätsdiagramm durch Aufruf der vordefinierten MEL-Operation `encrypt` modelliert. Im generierten Code wird hierfür die statische Methode `encrypt` der Klasse `EncDataSymm` aufgerufen. Diese ruft die Methode `encode` der Klasse `Coding` auf, die das zu verschlüsselnde Objekt `pd` in ein Byte-Array serialisiert und dieses als Ergebnis an die Klasse `EncDataSymm` zurückgibt. Für die Serialisierung wird die in Kapitel 6 erläuterte Kodierung verwendet. Im Anschluss daran wird die eigentliche Verschlüsselung durch Aufruf der Java Card Cryptography APIs [90] bzw. der Java Cryptography Architecture API [91] durchgeführt. Diese liefert ein Byte-Array zurück, das die verschlüsselten Daten enthält. Die Klasse `EncDataSymm` erzeugt ein neues Objekt vom Typ `EncDataSymm`, die das Byte-Array enthält und gibt es an die aufrufende Terminal- oder Smart Card-Klasse zurück. Das Vorgehen bei der Serialisierung sowie Deserialisierung der kryptographischen Daten entspricht an dieser Stelle dem Vorgehen bei der Serialisierung beim Versenden bzw. Empfang von Nachrichten. Die kryptographischen Funktionen des formalen Modells und die entsprechenden Implementierungen sind dann äquivalent, wenn die Serialisierung korrekt ist (siehe 10.3.6).

Operationen auf Listen

Assoziationsenden, die eine Multiplizität größer als Eins besitzen, werden in SecureMDD als Listen betrachtet. Für den Zugriff auf eine Liste gibt es in MEL vordefinierte Operationen. Ein Zugriff auf eine Liste ohne Verwendung dieser Operationen ist nicht möglich.

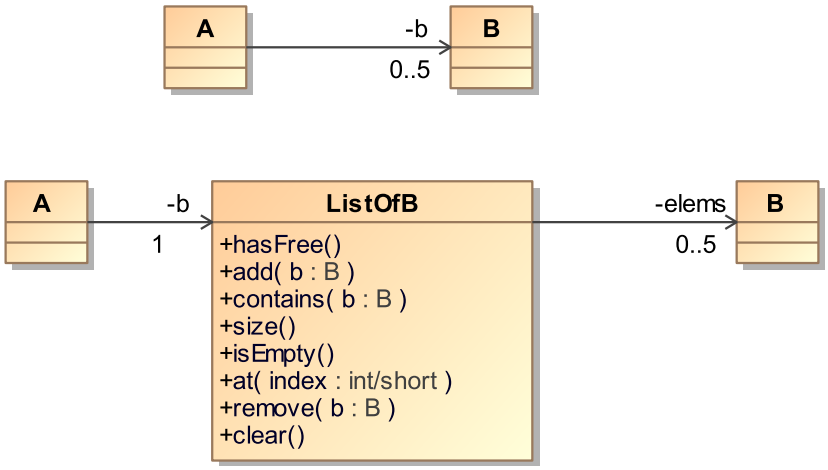

Abbildung 10.3.: Modellierung einer Liste in UML (oben) und Abb. auf den Code sowie das formale Modell (unten)

Abbildung 10.3 zeigt, wie eine in UML modellierte Liste (oben) im Code und in der formalen Spezifikation abgebildet wird (unten). Für die Klasse `B`, auf die die Assoziation zeigt, wird eine Containerklasse mit dem Namen `ListOfB` generiert, die die Listenelemente vom Typ `B` verwaltet. Die Klasse `A` referenziert nur noch diese Containerklasse, die die Methoden für den Zugriff auf die Liste bereitstellt.

Die Klasse `ListOfB` wird im Code auf eine Java-Klasse und im formalen Modell auf einen abstrakten Datentyp abgebildet. Die Liste selber ist im Code durch ein Array und im formalen Modell durch eine Liste repräsentiert. Die Liste im formalen Modell entsteht durch Aktualisierung der Spezifikation `list`. Da der Zugriff auf die Listen nur über die vordefinierten Operationen geschieht, ist dieser Unterschied nicht von Bedeutung. Für die Verfeinerung muss jedoch für jede Listenoperation gelten, dass die generierte Implementierung eine Verfeinerung der abstrakten Listenfunktion ist. Dies wurde durch ein sorgfältiges Vorgehen bei der Implementierung der Modelltransformationen sichergestellt.

10.3.4. Kopiersemantik

Objektidentitäten:

Die Sprache MEL ist objektorientiert, aber Objekte werden als Werte betrachtet, die keine eigene Identität haben. In Java dagegen gibt es Objektidentitäten, anhand derer zwei Objekte unterschieden werden können, d.h. zwei Objekte können unterschiedlich sein, auch wenn sie die gleichen Werte speichern. Die abstrakten Datentypen hingegen verwenden keine Identitäten

für die Daten. Die algebraisch spezifizierten Daten im formalen Modell sind gleich, wenn ihre Werte gleich sind.

Damit eine Verfeinerung der Datentypen möglich ist, müssen sich diese im Code so verhalten, als gäbe es die Objektidentitäten nicht. Aus diesem Grund wird für jeden nichtprimitiven Typ (d.h. alle Sicherheitsdatenklassen und alle im UML-Modell der Anwendung definierten Daten- und Nachrichtenklassen) eine `equals`-Methode generiert, die die Wertgleichheit zweier Objekte prüft. Wenn im UML-Aktivitätsdiagramm zwei Objekte mit dem ==-Operator auf Gleichheit getestet werden, wird dies im generierten Code in einen Aufruf der `equals`-Methode übersetzt. Dies entspricht dem Test zweier abstrakter Datentypen auf Gleichheit. Wegen der Zyklenfreiheit der Daten- und Nachrichtenklassen ist die Terminierung dieser Methoden garantiert. Aufgrund der Null-Freiheit des Codes können keine Java `NullPointerExceptions` auftreten.

Objektreferenzen:

Das Nicht-Vorhandensein von Objektidentitäten in MEL impliziert, dass es dort auch keine Referenzen auf andere Objekte gibt. Dies hat Einfluss auf die Semantik von Zuweisungen in MEL. Eine in MEL modellierte Zuweisung ist zum Beispiel `a := otherA`, die einer lokalen Variablen `a` den Wert eines Objekt mit Namen `otherA` vom selben Typ wie die lokale Variable zuweist. Da MEL keine Referenzen auf Objekte speichert, bedeutet diese Zuweisung, dass der Wert des Objekts `otherA` kopiert und die Kopie an die Variable `a` zugewiesen wird. Wird der Wert des Objekts `otherA` anschließend in einem weiteren MEL-Ausdruck verändert, hat dies keinen Einfluss auf den Wert der Variablen `a`. Dasselbe gilt, wenn nur einzelne Werte des in `a` gespeicherten Objekts verändert werden, z.B. `a.b := otherB`. Aus diesem Grund wird in dieser Arbeit davon gesprochen, dass MEL eine „Kopiersemantik“ besitzt.

Die algebraisch spezifizierten Datentypen im formalen Modell besitzen ebenfalls keine Identitäten und sind wertbasiert, d.h. bei einer Zuweisung eines Wertes an eine Variable hat die Variable den neuen Wert. Dies bedeutet, dass sich die ASM an dieser Stelle genauso verhält wie es die Sprache MEL vorsieht.

Bei einer Zuweisung in Java ist dies prinzipiell nicht so. Java hat eine Referenzsemantik, d.h. bei einer Zuweisung an ein Feld oder eine Variable wird nur die Referenz auf das zugewiesene Objekt gespeichert. Wird dieses nach der Zuweisung verändert, verändert sich somit auch der in dem Feld oder der Variablen gespeicherte Objektwert. Dass zwei Objekte Referenzen auf das gleiche Objekt speichern, wird auch als Sharing des Objekts bezeichnet. Ein Beispiel hierfür ist in Abb. 10.4 gegeben

Auch wenn die Klassen und Objekte als UML-Diagramm dargestellt sind, sind Java-Klassen sowie -Objekte gemeint. Die Klassen B und C besitzen beide eine gerichtete Assoziation auf die Klasse A. Die Klasse A muss in diesem Fall eine Datenklasse sein. B und C können Daten- oder Komponentenklassen (Smart Card oder Terminal) sein. In dem Beispiel gibt es eine Instanz der Klasse `C`, die eine Referenz auf ein Objekt `a` speichert. Weiterhin gibt es ein Objekt `b` vom Typ `B`, das eine Referenz auf ein anderes Objekt `otherA` vom Typ `A` speichert. Durch die Zuweisung `c.a := b.a`, würde nun die Instanz `c` ebenfalls eine Referenz auf das Objekt `otherA` speichern. Würde das Objekt `b` das Objekt anschließend verändern (zum Beispiel durch ein destruktives Update, bei dem das Feld `n` vom Typ `Nonce` verändert wird: `b.a.n := generateNonce()`), würde diese Änderung auch für `c` gelten.

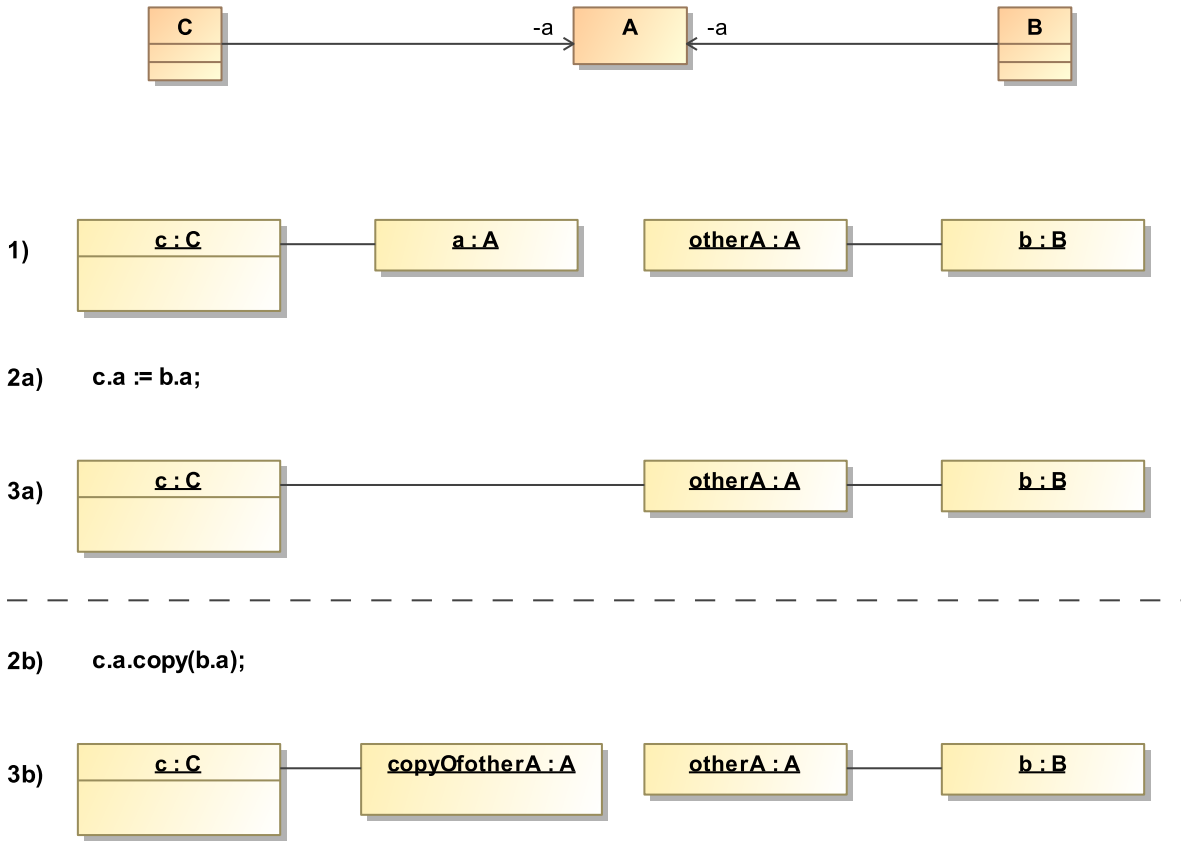

Abbildung 10.4.: Klassen A,B und C sowie Objektdiagramme

Erzwingen einer Kopiersemantik des Codes:

Damit der generierte Code sich bei Zuweisungen dennoch wie die ASM und MEL verhält, müssen die zugewiesenen Objekte rekursiv über alle Attribute und Assoziationen kopiert werden. Aus diesem Grund wird für jede Klasse automatisch eine `copy()`-Methode generiert, die rekursiv über alle Felder der Klasse die Objekte entsprechend kopiert. Wegen der Zyklenfreiheit der Daten- und Nachrichtenklassen terminieren die `copy()`-Methoden immer und aufgrund der Null-Freiheit des Codes können keine Java `NullPointerExceptions` auftreten.

Im Folgenden ist beispielhaft die generierte `copy()`-Methode für die Klasse `C` abgebildet:

```
public void copy(C from) {
   this.a.copy(from.a);}
```

Die Methode kopiert rekursiv die Felder des Objekts, das als Argument beim Aufruf der Methode übergeben wird. Dies ist das Objekt, das auf der rechten Seite der Zuweisung steht. Der MEL-Ausdruck `c.a := b.a` wird im generierten Code zu `c.a.copy(b.a)` (siehe 3b in Abb. 10.4) transformiert. Für Felder der Typen `int` bzw. `short`, `String` und `boolean` werden die Werte der Felder direkt kopiert. Für Byte-Arrays werden entsprechende Methoden zum Kopieren eines Arrays aufgerufen.

Zu beantworten ist noch, an welchen Stellen im generierten Code kopiert werden muss: Dies sind alle Stellen, an denen eine Zuweisung gemacht wird. Unter Zuweisung ist sowohl das Zuweisen eines neuen Objekts an ein Feld sowie an eine lokale Variable gemeint.

Mögliche Optimierungen:

Das Kopieren der Objekte bei Zuweisungen ist prinzipiell ineffizient in Bezug auf den benötigten Speicherplatz sowie die Laufzeit der Anwendung. Für die mit SecureMDD entwickelten Anwendungen, die in der Regel übersichtliche Datenstrukturen haben, ist dies jedoch kein großes Problem. Dennoch ist es möglich, an dieser Stelle den Code zu optimieren und nur an den Stellen zu kopieren, an denen es unbedingt notwendig ist.

Kritisch für das Sharing eines Objekts sind destruktive Updates. Dies bedeutet das Aktualisieren einzelner Werte eines Objekts. Im o.g. Beispiel geschieht dies durch den Ausdruck `b.a.n := generateNonce()`, der für das Feld `n` des Objekts `a` eine neue Nonce erzeugt. Referenzieren zwei Objekte das Objekt `a`, gilt diese Änderung somit für beide. Neben den destruktiven Updates, die explizit mit MEL in den Aktivitätsdiagrammen modelliert sind, gibt es noch implizite Zuweisungen, die durch Seiteneffekte in Operationen und vordefinierten Methoden entstehen. Ein Beispiel hierfür ist der ++-Operator, der den Wert eines Feldes oder einer Variablen inkrementiert und das Ergebnis an das Feld bzw. die Variable zuweist.

Wenn man sich den generierten Code genauer ansieht, stellt man fest, dass destruktive Updates nicht sehr häufig vorkommen. Bei den meisten Zuweisungen wird das zugewiesene Objekt im Anschluss gar nicht mehr verändert. In diesem Fall wäre es kein Problem, dass zwei Variablen Referenzen auf dasselbe Objekt speichern. An dieser Stelle gibt es somit die Möglichkeit, den generierten Code zu optimieren. Da der Code ohne Aufruf der `copy()`-Methode lesbarer und kürzer ist, erscheint diese Optimierung sinnvoll.

Die Lösung hierfür ist ein Algorithmus, der bei einer Zuweisung an eine lokale Variable nur dann kopiert, wenn im Anschluss an die Zuweisung noch ein destruktives Update stattfindet. Da lokale Variablen immer nur innerhalb eines Protokollschritts gültig sind, ist es möglich die Analyse für jeden Protokollschritt separat durchzuführen. Der Algorithmus muss also die Aktivitätsdiagramme durchgehen und für jede Zuweisung analysieren, ob in den folgenden MEL-Ausdrücken (bis zum Ende des Protokollschritts) ein destruktives Update geschieht. Ist dies der Fall, muss die Zuweisung in den Aufruf der `copy()`-Methode übersetzt werden. Im anderen Fall ist ein Sharing kein Problem und die Zuweisung kann in ein Java-Assignment übersetzt werden. Da die Felder einer Terminal- oder Smart Card-Klasse persistente Daten enthalten, muss bei Zuweisungen an diese Felder immer kopiert werden. Ansonsten können, über mehrere Protokollschritte und unterschiedliche Protokolle hinweg, Abhängigkeiten zwischen Objekten entstehen, die nur sehr schwer zu berechnen sind.

Der Entwurf und die Implementierung eines solchen Algorithmus sowie der Nachweis dessen Korrektheit sind noch offen. Die derzeitige Lösung ist jedoch ebenso korrekt, der Algorithmus wäre lediglich eine Optimierung.

10.3.5. Erzeugen eines neuen Objekts

Auf der Smart Card werden die Objekte, die bei der Deklaration von lokalen Variablen und für die Nachrichten benötigt werden sowie die entsprechenden Unterobjekte bereits bei Erzeugung

der Smart Card-Klasse (d.h. des Applets) erstellt und von einem Objektmanager verwaltet. Damit die Verwendung des Objektmanagers die Äquivalenzbeziehung nicht beeinflusst, muss dieser einige Kriterien erfüllen:

- Der Objektmanager darf jedes Objekt zu jedem Zeitpunkt maximal einmal herausgegeben haben. Wird ein verwaltetes Objekt zweimal in einem Protokollschritt herausgegeben, kommt es zum Objektsharing und einer Verletzung der Kopiersemantik. Die Herausgabe ist folgendermaßen gelöst: Die verwalteten Objekte eines Typs sind in einem Array gespeichert, ein Index zeigt auf das zuletzt herausgegebene Objekt. Vor Herausgabe eines Objekts wird dieser Index hochgezählt, d.h. er zeigt dann auf das nächste, noch freie Objekt.
- Am Ende jedes Protokollschritts bzw. vor Beginn eines neuen Schritts müssen alle in diesem Schritt herausgegebenen Objekte wieder an den Objektmanager „zurückgegeben“ werden. Dies geschieht durch Rücksetzen der Indexe auf den Wert null in dem Moment, in dem die Smart Card eine neue Nachricht als APDU empfängt (und bevor diese Nachricht deserialisiert wird). Eine Rückgabe ist nur möglich, wenn kein Feld der Smart Card-Klasse eine Referenz auf ein verwaltetes Objekte speichert. Dies ist dadurch sichergestellt, dass bei Zuweisung an ein Feld das zugewiesene Objekt immer kopiert wird. Außerdem dürfen die verwalteten Nachrichtenobjekte nicht mehr in Verwendung sein. Dies ist der Fall, weil Nachrichten nicht gespeichert werden können und sowohl die empfangene als auch die gesendete Nachricht am Ende eines Schrittes, d.h. konkreter nach der Serialisierung des zu sendenden Nachrichtenobjekts, nicht mehr benötigt werden.
- Der Objektmanager muss immer ausreichend viele Objekte vorrätig haben, um zu jedem Zeitpunkt einen beliebigen Protokollschritt ausführen zu können. Dies ist notwendig, weil der Angreifer beliebige Nachrichten an eine Smart Card schicken und somit auch Protokollschritte außer der Reihe initiieren kann. Die Berechnung der benötigten Objekte erfolgt anhand der Aktivitätsdiagramme. Es wird berechnet, wie viele Objekte jedes Typs maximal in einem Schritt benötigt werden und diese Anzahl im Objektmanager bereitgestellt. Da am Ende eines Protokollschritts alle in diesem Schritt verwendeten Objekte wieder zur Verfügung stehen und ein Protokollschritt (für den in dieser Arbeit betrachteten Angreifer) als atomar angesehen werden kann, stehen für alle Protokollschritte jederzeit ausreichend Objekte zur Verfügung.
- Durch die Verwendung des Objektmanagers kommt es temporär, für die Objekte, die aktuell herausgegeben wurden, zu einem Objektsharing. Dies ist der Fall, weil der Objektmanager weiterhin eine Referenz auf jedes Objekt speichert, das er herausgegeben hat. Da der Objektmanager aber weder lesend noch schreibend auf die Objekte und ihre Felder zugreift, ist das Sharing an dieser Stelle kein Problem.

Die Korrektheit des generierten Codes für den Objektmanager wurde durch Verwendung in mehreren Fallstudien sowie Codereviews überprüft. Durch die Einhaltung der o.g. Punkte ist die Verwendung des Objektmanagers äquivalent zu der Erzeugung entsprechender Elemente der abstrakten Datentypen.

10.3.6. Kommunikation zwischen Terminal und Smart Card

Senden und Empfangen von Nachrichten

Das Senden einer Nachricht umfasst im generierten Code die Serialisierung der Nachricht in ein Byte-Array und das Versenden dieses Arrays. Für den Versand werden (teilweise native) Methoden der Java Card-API verwendet. Beim Empfang einer Nachricht wird die Methode `process(APDU apdu)` aufgerufen, die die Nachricht zunächst deserialisiert und anschließend verarbeitet, d.h. den entsprechenden Protokollschritt ausführt. Die Kommunikation ist synchron realisiert, d.h. ein Terminal wartet nach Senden einer Nachricht an eine Smart Card auf die Antwortnachricht.

Im formalen Modell ist das Senden und Empfangen von Nachrichten über Inboxen realisiert. Jeder Agent besitzt für jeden seiner Ports eine Inbox. Dies ist eine Liste unbegrenzter Länge, in der Nachrichten gespeichert werden können. Das Senden einer Nachricht ist als das Einfügen dieser in die entsprechende Inbox des empfangenden Agenten realisiert. Der Empfang einer Nachricht entspricht der Entnahme dieser Nachricht aus der Inbox. Nach Entnahme einer Nachricht wird diese verarbeitet, d.h. der entsprechende Protokollschritt durchgeführt. In einem Protokolllauf, in dem der Angreifer keine Nachrichten generiert und an die Agenten sendet, befindet sich zu jedem Zeitpunkt in einer Inbox maximal eine Nachricht. Die Kommunikation im formalen Modell ist asynchron realisiert. Dadurch bedingt sind im formalen Modell Abläufe möglich, die im generierten Code nicht möglich sind. Umgekehrt gilt jedoch, dass jeder Ablauf auf Codeebene auch im formalen Modell möglich ist. Daher gilt die Refinementbeziehung. Die Serialisierung und Deserialisierung, die im Code bei der Kommunikation zwischen einem Terminal und einer Smart Card sowie vor der Durchführung kryptographischer Operationen stattfindet, ist im formalen Modell nicht abgebildet. Dieser Aspekt wird deshalb im Folgenden genauer betrachtet.

(De-)Serialisierung von Nachrichtenobjekten

Durch die (De-)Serialisierung dürfen keine zusätzlichen Angriffsmöglichkeiten entstehen. Auf Implementierungsebene hat der Angreifer jedoch die Möglichkeit, einzelne Bits in dem übertragenen Byte-Array zu verändern, das übertragene Array zu verlängern und zu verkürzen oder das gesamte Array auszutauschen. Da die Abbildung von Objekten in Byte-Arrays nicht surjektiv ist, können durch diese Manipulation Byte-Arrays entstehen, für die es keine gültige Objektrepräsentation gibt und die beim Dekodieren des Arrays zurückgewiesen werden müssen. Dieser Zusammenhang ist in Abbildung 10.5 dargestellt.

Die kodierte Nachricht `ba` wird zunächst von der Klasse `Coding` durch Aufruf der Methode `decode` deserialisiert. Ist die Kodierung gültig (`ok`), wird die Methode `processMsgName()` aufgerufen, die den entsprechenden Protokollschritt ausführt. Tritt bei der Deserialisierung ein Fehler auf, ist das empfangene Byte-Array `ba` keine gültige Repräsentation eines Nachrichtenobjekts. Es ist davon auszugehen, dass ein Angreifer dieses Byte-Array manipuliert hat oder ein Übertragungsfehler aufgetreten ist. In diesem Fall (`!ok`) wird die (serialisierte) Nachricht abgewiesen, d.h. kein Protokollschritt ausgeführt. Dies bedeutet, dass nur valide Nachrichtenobjekte von den Smart Cards und Terminals verarbeitet werden, alle ungültigen Byte-Arrays werden erkannt und verworfen.

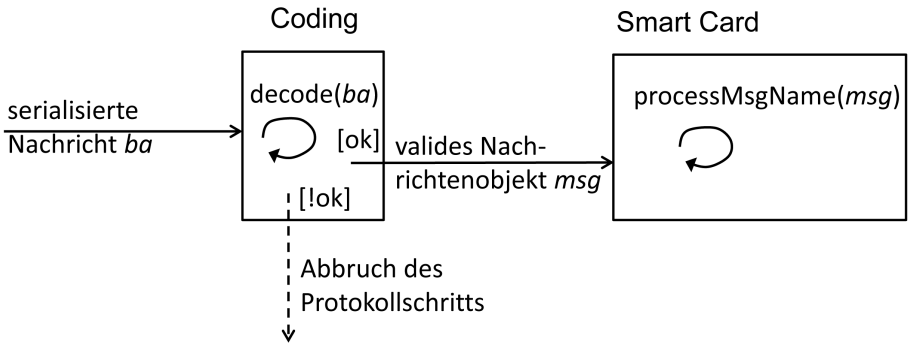

Abbildung 10.5.: Deserialisierung einer kodierten Nachricht nach dem Empfang

Zusätzlich gibt es jedoch eine Angriffsmöglichkeit auf Implementierungsebene, die ebenfalls beachtet werden muss: Durch Verlängern einer kodierten Nachricht (insbesondere eines kodierten Byte-Arrays) sowie durch Manipulation der Längenangabe eines Byte-Arrays (d.h. der zwei Bytes, die die Länge eines kodierten Byte-Arrays speichern), kann es beim Dekodieren der Nachricht zu einem Übertreten der Array-Grenzen kommen. In einer fehlerhaften Implementierung würde die im Längenfeld angegebene Anzahl von Bytes kopiert werden, ohne zu überprüfen, dass Quell- und Zielarray lang genug dafür sind. Die Implementierung der Deserialisierung stellt sicher, dass diese Art von Angriff nicht möglich ist (siehe Abschnitt 10.3.9).

Durch die Implementierung der (De-)Serialisierungsschicht ist somit sichergestellt, dass alle ungültigen Kodierungen erkannt und herausgefiltert werden. Weiterhin entstehen durch diese Schicht keine zusätzlichen Sicherheitslücken. Im formalen Modell ist das Senden eines manipulierten Byte-Arrays, d.h. einer ungültigen Serialisierung einer Nachricht an eine Komponente dadurch adäquat abgebildet, dass der Angreifer jederzeit eine ungültige Nachricht (`InvalidMessage`) in eine Inbox schreiben kann. Wird eine solche Nachricht aus der Inbox entnommen, bricht die aktuelle ASM-Regel ab, d.h. es wird die Prozedur STOPSTEP aufgerufen und kein Protokollschritt ausgeführt.

Es kann außerdem passieren, dass durch Manipulation des Byte-Arrays durch einen Angreifer ein anderes gültiges Nachrichtenobjekt entsteht. Auf Implementierungsebene bedeutet dies, dass jeder Protokollschritt prinzipiell zu jeder Zeit ausführbar sein muss. Dies ist durch den Objektmanager sichergestellt, der für jeden Protokollschritt ausreichend viele Objekte bereithält. Tauscht der Angreifer während der Übertragung eine Nachricht aus, wird diese deserialisiert und der entsprechende Protokollschritt kann ausgeführt werden. Auf formaler Ebene ist der Austausch einer Nachricht durch den Angreifer ebenfalls möglich. Da dort beliebig viele Elemente eines Datentyps erzeugt werden können, kann auch hier jeder Protokollschritt zu jeder Zeit ausgeführt werden.

Zusätzlich zu den Sicherheitslücken, die theoretisch durch die Verwendung der (De-)Serialisierung entstehen können, ist es essentiell, dass die (De-)Serialisierung korrekt implementiert ist. Dabei sind zwei Aspekte gesondert zu betrachten:

- Eine essentielle Anforderung an die Serialisierung ist ihre Injektivität. Diese stellt sicher, dass eine Byte-Array Repräsentation auf Empfängerseite eindeutig einem Nachrichtenobjekt zugeordnet werden kann. Hierfür ist es notwendig, die Information über den Typ der Nachrichten und ihren Attributen und Assoziationsenden mit in dem Byte-Array zu speichern. Wäre dies nicht der Fall, wäre es nicht möglich, z.B. einen serialisierten String von einem Geheimnis (vom Typ `Secret`) zu unterscheiden. Die Serialisierung verwendet aus diesem Grund Typflags, die mit in den Byte-Arrays gespeichert werden.
- Der primitive Datentyp `Number` wird auf Terminalseite in Integer übersetzt, während für den Smart Card-Code der Typ `short` verwendet wird. Wird ein Integer vom Terminal an die Karte geschickt, muss dieser dort während der Deserialisierung in einen `short`-Wert umgewandelt werden. Dabei kann es passieren, dass der Integer zu groß ist, um ihn als `short`-Wert abzuspeichern. In diesem Fall wirft die Implementierung der Deserialisierung eines Integers eine SecureMDD-Exception und der Protokollschritt, der auf den Empfang der Nachricht folgen würde, wird nicht ausgeführt. Dieses Exceptionverhalten muss adäquat im formalen Modell nachgebildet werden. Aus diesem Grund wird auch dort beim Empfang einer Nachricht, die einen Integer enthält, zu Beginn eines Protokollschritts mithilfe des Prädikats `shortOverflow` überprüft, ob der Wert zu groß ist, um ihn als `short` zu speichern. In diesem Fall wird die Prozedur STOPSTEP aufgerufen (siehe Abschnitt 10.3.7), die den Protokollschritt beendet.

In [67] beschreiben Grandy et al. eine für die Entwicklung von Sicherheitsprotokollen geeignete verifizierte Transformationsschicht. Diese Kodierungsschicht, die ebenfalls eine TLV-Kodierung verwendet, ist ähnlich zu der in dieser Arbeit realisierten (De-)Serialisierungsschicht. Anstatt der anwendungsspezifischen Daten- und Nachrichtentypen verwendet Grandy jedoch ein anwendungsunabhängiges Datenformat. In [67] wird die Transformationsschicht als korrekt nachgewiesen und es wird gezeigt, dass die in der Implementierung verwendeten Datentypen eine Verfeinerung der abstrakten Datentypen des formalen Modells sind.

In diesem Abschnitt wurde begründet, dass die Kommunikation zwischen einem Terminal und einer Smart Card adäquat im formalen Modell abgebildet ist. Die in der Implementierung verwendete Serialisierungsschicht ist so realisiert, dass der Angreifer in der Realität keine Angriffsmöglichkeiten hat, die nicht auch im formalen Modell zugelassen sind. Die Implementierung der Kommunikation ist somit eine Verfeinerung der Kommunikationsstruktur des formalen Modells.

10.3.7. Abbruch eines Protokollschritts

Dem Abbruch eines Protokollschritts geht immer das Auftreten eines Fehlers voraus. Es gibt dabei zwei Arten von Fehlern, explizite sowie implizite Fehler. Der Begriff „expliziter Fehler“ wird verwendet, weil dieser explizit im UML-Modell, durch einen UML-`FlowFinalNode`, angegeben wird. Dies ist zum Beispiel sinnvoll, wenn eine empfangene Nonce nicht mit der erwarteten Nonce übereinstimmt. Ein `FlowFinalNode` hat keine ausgehenden Kanten, d.h. bei Erreichen dieses Knotens wird der Protokollschritt nicht weiter ausgeführt. Implizite Fehler dagegen kommen im UML-Modell nur durch den Aufruf einer vordefinierten MEL-Methode, die eine Exception werfen kann, vor. Beispiele sind das Entschlüsseln eines Dokuments mit dem falschen Schlüssel oder das Hinzufügen eines Elements zu einer bereits vollen Liste.

Im Folgenden ist die Übersetzung des Auftretens eines expliziten Fehlers erläutert. Ein Beispiel für einen impliziten Fehler sowie die Abbildung in Code und formales Modell ist die Addition zweier Zahlen und die Behandlung eines möglichen Over- oder Underflows. Dies wurde bereits in Abschnitt 10.3.1 beschrieben.

Abbruch bei Auftreten eines expliziten Fehlers

Ein UML-`FlowFinalNode` wird im Java-Code in den Aufruf der Methode `stop()` übersetzt, d.h. an den Stellen, an denen im UML-Modell der `FlowFinalNode` erreicht wird, wird im Code die `stop()`-Methode aufgerufen, die eine `ISOException` wirft.

Der generierte Smart Card-Code sieht folgendermaßen aus:

```
public void stop(){
ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);}
```

Auf einer Smart Card beginnt ein Protokollschritt immer mit dem Empfang einer Nachricht von einem Terminal und endet mit dem Senden einer Rückantwort. Wenn im Code bei Ausführung des Schritts kein Fehler auftritt, enthält die Rückantwort das Statuswort „Ok“ sowie eine Antwortnachricht vom Typ `Message`. Wird auf der Karte ein Fehler, d.h. eine `ISOException`, geworfen, wird an dieser Stelle ein Fehler-Statuswort und keine Antwortnachricht zurückgesendet. Im Anschluss daran kann die Smart Card weitere Nachrichten von dem Terminal empfangen und somit erneut Protokollschritte ausführen.

Der entsprechende Terminal-Code (der Klasse `DepositMachine`) ist nachfolgend abgebildet:

```
protected void stop() throws TerminalException{
throw new TerminalException("DepositMachine");}
```

Im Terminal-Code wird bei Auftreten eines Fehlers eine SecureMDD-Exception (`TerminalException`) geworfen und somit der aktuelle Protokollschritt abgebrochen. Im Anschluss daran kann das Terminal weitere Nachrichten (von einem Benutzer) empfangen und somit weitere Protokollschritte ausführen.

Eine Besonderheit auf Terminalseite ist das Empfangen des Fehler-Statuswortes einer Smart Card, nachdem dort ein Fehler aufgetreten ist. Der Grund hierfür ist, dass das Terminal nach Versenden einer Nachricht synchron auf die Rückantwort der Karte wartet. Ein Protokollschritt auf der Smart Card und der nachfolgende Terminal-Protokollschritt hängen somit eng zusammen. Empfängt das Terminal die Rückantwort (mit einem gültigen Nachrichtenobjekt), ruft es die Methode auf, die den nächsten Protokollschritt auf Terminalseite ausführt. Empfängt das Terminal einen Fehlercode als Rückantwort, wird der nachfolgende Terminal-Schritt unterbunden, d.h. nicht ausgeführt. In diesem Fall kann das Terminal aber weitere Nachrichten (von einem Benutzer) empfangen und somit auch weitere Protokollschritte ausführen.

In der ASM wird an den Stellen, an denen im Code ein Fehler auftritt, die Prozedur `STOPSTEP` aufgerufen. Ihre Deklaration ist im Folgenden angegeben:

```
STOPSTEP : STOPSTEP(exception_occurred)
{exception_occurred := true};
```

Die Prozedur setzt die Variable `exception_occurred` auf `true`. Damit ist die aktuelle ASM-Regel beendet und die ASM-Regel (mit Namen `ASM`) wird im Anschluss erneut aufge-

rufen, um nichtdeterministisch den nächsten Schritt auszuwählen oder zu terminieren. Das bedeutet, dass auch im formalen Modell nach Auftreten eines Fehlers der aktuelle Protokollschritt beendet wird und im Anschluss daran weitere ASM-Schritte ausgeführt werden können. Im Falle des Auftretens eines Fehlers auf einer Smart Card wird der nachfolgende Protokollschritt eines Terminals im formalen Modell dadurch unterbunden, dass der Smart Card-STEP, in dem der Fehler aufgetreten ist, keine entsprechende Nachricht in die Terminal-Inbox geschrieben hat. Die boolesche Variable `exception_occurred` wird für Subdiagrammaufrufe benötigt. Dieser Spezialfall wird im folgenden Abschnitt betrachtet.

Weitergabe eines Fehlers bei Subdiagrammaufrufen

Tritt in einem Subdiagramm ein (impliziter oder expliziter) Fehler auf, verhält sich der Java-Code so wie oben beschrieben, d.h. es wird die `stop()`-Methode aufgerufen, eine entsprechende Exception geworfen und diese an die Methode, die das Subdiagramm aufgerufen hat, weitergegeben. Diese Methode bricht dann ab, d.h. der Protokollschritt wird nicht weiter ausgeführt.

Im formalen Modell wird an entsprechender Stelle im Subdiagramm ebenfalls die Prozedur `STOPSTEP` aufgerufen, d.h. das Subdiagramm wird nicht weiter ausgeführt. Subdiagramme werden im Formalen in Prozeduren übersetzt. Damit die aufrufende ASM-Regel erfährt, dass in der aufgerufenen Prozedur ein Fehler aufgetreten ist, muss ihr dies explizit mitgeteilt werden. Hierfür wird die Variable `exception_occurred` verwendet, die in der `STOPSTEP`-Prozedur auf `true` gesetzt wird. Nach dem Aufruf einer Prozedur wird deshalb in der ASM-Regel überprüft, ob dieses Flag gesetzt wurde. Der entsprechende Ausschnitt aus der formalen Spezifikation ist im Folgenden dargestellt.

```
SUBDIAGRAMMAUFRUF(exception_occurred,..);
if (not exception_occurred)
then{..}
```

Wurde das Flag `exception_occurred` gesetzt, wird die Ausführung der ASM-Regel beendet. Wurde das Flag nicht auf `true` gesetzt, wird der Protokollschritt weiter ausgeführt (in dem Spezifikationsausschnitt durch .. dargestellt).

Da der Aufruf einer Prozedur (sowie auch eine Fallunterscheidung) in der ASM nicht an beliebiger Stelle vorkommen kann, müssen sich in der Modellierung alle MEL-Ausdrücke, die eine Exception werfen können, auf oberster Ebene (d.h. nicht geschachtelt in anderen Ausdrücken) oder auf der rechten Seite einer Zuweisung befinden. Außerdem können sie als Initialausdruck bei der Deklaration lokaler Variablen angegeben werden. Nicht zugelassen sind solche Ausdrücke zum Beispiel in UML-`Guards` einer Fallunterscheidung oder innerhalb eines `Create`-Ausdrucks zum Erzeugen eines neuen Objekts. Dies wird bei der Validierung der UML-Modelle überprüft.

In diesem Abschnitt wurde erläutert, weshalb das Auftreten expliziter und impliziter Fehler sowie das Auftreten eines Fehlers innerhalb eines Subdiagramms adäquat im formalen Modell abgebildet sind. Weitere Fehlertypen gibt es nicht. Aus diesem Grund gilt bezüglich des Abbruchs eines Protokollschritts die Äquivalenzbeziehung.

10.3.8. Kein Auftreten von Null

MEL sowie die formale Spezifikation sind `Null`-frei, d.h. es gibt kein `Null`-Literal. Im Java-Code können Felder und Variablen theoretisch `Null` sein, d.h. sie verweisen auf kein Objekt. Um dennoch eine Äquivalenz zwischen dem Code und dem formalen Modell herzustellen, erzwingt die Codegenerierung eine `Null`-Freiheit auch für den Code. Dies wird durch die Umsetzung der folgenden zwei Punkte erreicht:

- wie im formalen Modell sind im Code alle Felder und lokale Variablen initialisiert, d.h. bei Erzeugung ungleich `Null`. Für lokale Variablendeklarationen muss deshalb ein Initialwert angegeben werden. Die Felder, denen nicht explizit ein Initialwert gegeben wird, bekommen Defaultwerte, z.B. den Wert `0` für Felder vom Typ `Number` oder einen Defaultschlüssel für Felder vom Typ `SymmKey`. Dies ist rekursiv für alle Felder realisiert, d.h. auch alle Unterobjekte sind initialisiert.
- der generierte Code enthält an keiner Stelle das `Null`-Literal. Dies bedeutet, dass auch zur Laufzeit keine Felder oder Variablen `Null` sein und Methoden `Null` nicht als Rückgabewert haben können.

10.3.9. Keine Laufzeitfehler

In MEL sowie im formalen Modell können keine Laufzeitfehler auftreten. Im Java-Code hingegen ist das Auftreten von `RuntimeExceptions` möglich. Die Laufzeitfehler, die bei Ausführung des generierten Codes theoretisch auftreten können, sind `NullPointerExceptions`, `ArrayIndexOutOfBoundsExceptions`, `ClassCastExceptions`, `ArithmeticExceptions` sowie `CryptoExceptions` (auf der Smart Card).

Die Modelltransformationen sind so implementiert, dass der generierte Code keine Laufzeitfehler wirft. Für jede der oben genannten Exceptionklasse wird im Folgenden erläutert, an welchen Stellen im Code dieser Fehlertyp theoretisch auftreten kann und wie die Lösung aussieht, damit bei Ausführung des entsprechenden Codes keine Laufzeitfehler auftreten können.

`NullPointerExceptions` können aufgrund der `Null`-Freiheit des generierten Java-Codes nicht auftreten (siehe Abschnitt 10.3.8).

`ArrayIndexOutOfBoundsExceptions` können im generierten Code nur an wenigen Stellen auftreten. Diese werden nachfolgend betrachtet:

- Eine Liste wird im Code durch ein Array (und eine ArrayList für Listen unbeschränkter Länge) repräsentiert, auf das über vordefinierte Methoden zugegriffen werden kann. Zwei dieser Methoden können theoretisch eine `ArrayIndexOutOfBoundsException` werfen:
  - die Methode `add(Element e)`, wenn die Länge der Liste beschränkt und im Moment des Aufrufs bereits voll ist.
  - die Methode `at(int/short index)`, wenn sich an der Stelle in der Liste, die durch den Index angegeben ist, kein gültiges Element befindet.

Als Lösung wird zunächst angefragt, ob die Liste noch freie Plätze bzw. der Index gültig ist. Ist dies nicht der Fall, wird durch Aufruf der Methode `stop()` eine SecureMDD-Exception geworfen. Dies ist beispielhaft für die Methode `add` illustriert:

```
public void add(Element e) {
if(hasFree()) {
   elems[index++].copy(e);}
else {
   stop();}}
```

In der formalen Spezifikation wird in der ASM vor Aufruf der Funktion `add` (ebenfalls mithilfe der Funktion `hasFree()`) überprüft, ob die Liste noch freie Elemente hat. Ist dies der Fall, wird die Funktion `add` aufgerufen, anderenfalls wird die Prozedur `STOPSTEP` aufgerufen. Dieses Vorgehen ist analog zu der Behandlung des Over- bzw. Underflows bei arithmetischen Operationen (siehe Abschnitt 10.3.1) und äquivalent zum Werfen einer SecureMDD-Exception im Code.

- Außerdem werden bei der Deserialisierung von Objekten Byte-Arrays verwendet. Auf Terminalseite werden diese bei Bedarf mit passender Länge neu erzeugt. Auf der Karte werden Byte-Arrays aufgrund des geringen Speicherplatzes wiederverwendet. Dabei kann an zwei Stellen theoretisch eine `ArrayIndexOutOfBoundsException` auftreten. Erstens beim Übertreten der Arraygrenzen des Arrays `decodebuffer`, in das die kodierten Daten (nach dem Empfang) geschrieben wurden. Die Länge dieses Arrays ist ausreichend lang, um die Byte-Array-Repräsentation jedes Objekts, dass während der Protokollläufe serialisiert werden kann, zu speichern. Hierfür wird die Länge des Byte-Arrays während der Codegenerierung anhand der UML-Klassendiagramme berechnet. Es kann allerdings passieren, dass ein Angreifer die kodierte Nachricht verlängert (zum Beispiel durch Hochsetzen des Längenwertes eines kodierten Byte-Arrays und Anhängen der entsprechenden Bytes) und die kodierte Nachricht dann zu lang für die Speicherung im Array `decodebuffer` ist. Zweitens kann es passieren, dass es in der Methode `decodeByteArray` beim Schreiben des dekodierten Byte-Arrays in das Array `into` zu einem Übertreten der Arraygrenzen kommt. Die Ursache hierfür ist ebenfalls eine vorangegangene Manipulation einer kodierten Nachricht während des Übertragens an den Empfänger.

  Der generierte Code implementiert, wenn möglich, entsprechende Tests, die sicherstellen, dass die Arraygrenzen beim Dekodieren von Daten in Arrays nicht überschritten werden. Da dies jedoch nicht an allen Stellen möglich ist, fangen die `decode`-Methoden die `ArrayIndexOutOfBoundsExceptions` ab und werfen eine ISOException. Diese beendet den Protokollschritt, d.h. die empfangenen ungültigen Daten werden nicht an die Smart Card-Komponente weitergeleitet. Das Verhalten des Codes ist somit diesbezüglich äquivalent zu dem Verhalten des formalen Modells.

- Kryptographische Daten wie z.B. Signaturen und Hashwerte sind im Code ebenfalls als Byte-Arrays, in Objekten der Sicherheitsdatentypen `SignedData`, `HashedData` usw. gespeichert. Hashwerte und Signaturen besitzen eine fixe Länge. Die entsprechenden Byte-Arrays sind somit ausreichend lang (20 Byte für eine SHA1-Hashwert und 128 Byte für RSA-SHA1-Signaturen). Verschlüsselte Daten werden ebenfalls als Byte-Array gespeichert. Die Länge dieses Byte-Arrays entspricht dabei der Länge des Byte-

Arrays, das die Klartextdaten enthält (plus evtl. einigen Padding-Bytes). Die Länge wird deshalb aus dem längsten Klartextdatum, das während einem Protokolllauf verschlüsselt wird, berechnet. Die Berechnung geschieht durch Analyse der im Klassendiagramm definierten Nachrichten- und Datenklassen. Da die Länge der benötigten Byte-Arrays vor Generierung des Codes bestimmt wird, kann während der Laufzeit keine `ArrayIndexOutOfBoundsException` auftreten.

- Strings werden auf Smart Card-Seite durch Byte-Arrays repräsentiert. Die Maximallänge eines Strings ist konfigurierbar, momentan ist sie auf 20 Zeichen begrenzt. Die Einhaltung dieser Länge (auf Karten- sowie Terminalseite) wird während der Validierung des UML-Modells überprüft. Wird ein String in einer Benutzernachricht an das Terminal gesendet, darf dieser ebenfalls nicht länger als die Maximallänge sein. Auch dies wird überprüft. Ein Byte-Array ist ausreichend lang (d.h. 20 Byte), um die ASCII-Repräsentation des Strings zu speichern. An dieser Stelle kann somit ebenfalls keine `ArrayIndexOutOfBoundsException` auftreten.

Eine `ClassCastException` kann auftreten, wenn ein Byte-Array entschlüsselt, anschließend deserialisiert und dann weiterverwendet, d.h. zum Beispiel in einer lokalen Variablen gespeichert, wird. Die Methode `decrypt` der Klasse `EncDataSymm` liefert ein Objekt vom Typ `PlainData`, der noch in einen konkreten Typ gecastet werden muss. Durch die Codegenerierung und die der Generierung vorgeschalteten Validierung ist sichergestellt, dass der Cast an dieser Stelle möglich ist und keine Exception geworfen wird.

Eine `ArithmeticException` kann bei der Division durch null auftreten. Die Lösung ist auf die gleiche Weise wie der Umgang mit Over- bzw. Underflows bei der Durchführung arithmetischer Operationen realisiert (siehe Abschnitt 10.3.1). Im generierten Code wird vor Ausführen der Operation überprüft, ob der Divisor gleich null ist und in diesem Fall eine SecureMDD-Exception geworfen. Das formale Modell führt diesen Test ebenfalls vor der Durchführung der Division auf und ruft ggf. die Prozedur `STOPSTEP` auf.

Die Implementierung der in MEL vordefinierten Methoden auf Smart Card-Seite kann theoretisch `CryptoExceptions` werfen. Dies ist z.B. beim Ver- und Entschlüsseln eines Datums oder dem Bilden eines Hashwerts oder einer digitalen Signatur möglich. Die Implementierung sowie die vor der Codegenerierung stattfindende Validierung stellen jedoch sicher, dass diese Exceptions bei Ausführung des Codes nicht auftreten können. Ein Beispiel hierfür ist das Auftreten einer `NoSuchAlgorithm` Exception bei Verwendung eines falschen, d.h. nichtexistierenden Verschlüsselungsalgorithmus (dieser muss als Byte übergeben werden, die möglichen Algorithmen sind als Konstanten vom Typ Byte definiert). In der Implementierung wird jedoch ein gültiges Byte, nämlich das für die Konstante „ALG_DES_CBC_NOPAD“ (DES-Verschlüsselung im CBC-Modus ohne Padding) verwendet. Eine `CryptoException` kann somit an dieser Stelle nicht auftreten.

Auf Terminalseite sind `GeneralSecurityExceptions` das Java-Äquivalent zu den `CryptoExceptions` auf Smart Card-Seite. Dies sind keine Laufzeitfehler und werden in den entsprechenden Methoden gefangen. Auch hier gilt, dass die Generierung und die Validierung sicherstellen, dass diese Exceptions bei Ausführung des Codes nicht geworfen werden.

Somit gilt, dass die Modelltransformationen und der erzeugte Java-Code so implementiert wurden, dass der generierte Code keine Laufzeitfehler wirft. An einigen Stellen wird statt des Laufzeitfehlers eine SecureMDD-Exception geworfen, die jedoch adäquat im formalen Modell

abgebildet ist. Aus diesem Grund haben Laufzeitfehler keinen Einfluss auf die Äquivalenzbeziehung.

Zusammenfassend lässt sich somit sagen, dass der generierte Quellcode eine Verfeinerung des ebenfalls generierten formalen Modells ist. Diese Verfeinerungsbeziehung gilt für alle mit dem SecureMDD-Ansatz entworfenen Smart Card-Anwendungen. Generell übertragen sich die auf dem formalen Modell bewiesenen Sicherheitseigenschaften jedoch auch dann nicht automatisch auf die Implementierung, wenn diese eine Verfeinerung des abstrakten Modells ist. Damit sich die Eigenschaften übertragen, muss für die mit dem SecureMDD-Ansatz entwickelten Anwendungen bzw. die Produktivumgebung gelten, dass ein Angreifer keinen Zugriff auf den Speicher und die internen Abläufe der Komponenten, d.h. die Smart Cards und Terminals hat („Black Box"-Sicht). Dies bedeutet zum Beispiel, dass der Angreifer keine Schadsoftware auf den Terminals ausführen kann. Diese bereits in Abschnitt 2.2 diskutierte Annahme wird auch im formalen Modell vorausgesetzt und begründet sich damit, dass der SecureMDD-Ansatz keine Lösungsmöglichkeiten für den Schutz der Terminals vor dem Zugriff Dritter bietet, sondern die Entwicklung sicherer kryptographischer Protokolle im Vordergrund steht. Eine weitere Voraussetzung dafür, dass sich die bewiesenen Eigenschaften auf die Implementierungsebene übertragen ist die korrekte Implementierung der (De-)Serialisierungsschicht. Insbesondere gilt hier, dass ungültige Serialisierungen von Nachrichten (die zum Beispiel von einem Angreifer erzeugt und versendet wurden) während der Deserialisierung erkannt und verworfen werden (siehe Abschnitt 10.3.6, Seite 255). Für die mit SecureMDD entwickelten Anwendungen gilt somit, dass sich die bewiesenen Sicherheitseigenschaften auch auf den generierten Quellcode übertragen.

10.4. Verwandte Arbeiten

Für alle Smart Card-Anwendungen, die mit dem SecureMDD-Ansatz entwickelt werden, gilt, dass die auf formaler Ebene bewiesenen Sicherheitseigenschaften ebenfalls auch für den generierten Quellccode gelten. Es gibt andere Ansätze, die die Sicherheit einer konkreten Implementierung verifizieren. Mir ist jedoch kein anderer Ansatz bekannt, der sicheren (Java Card-)Quellcode generiert, d.h. bei dem Sicherheitseigenschaften auf einem abstrakten formalen Modell bewiesen werden, die sich automatisch und für alle mit dem Ansatz entwickelten Anwendungen auf den generierten Code übertragen. Im Folgenden werden drei verschiedene Techniken, mit denen die Sicherheit einer Implementierung nachgewiesen werden kann, vorgestellt. Zunächst sind dies die klassischen Ansätze zur Verfeinerung. Die Theorien zur Verfeinerung werden jedoch in der Literatur eher auf theoretischer Ebene betrachtet. Die einzige Arbeit, in der eine Implementierung (einer konkreten Anwendung) als Verfeinerung eines abstrakten Modells nachgewiesen wird, ist die Arbeit von Grandy [64], die bereits am Anfang dieses Kapitels vorgestellt wurde. Das zweite Forschungsgebiet, das sich mit der Sicherheit einer Implementierung beschäftigt, ist die Quellcodeverifikation (z.B. [13,33,40]). Auch hier gibt es bisher keinen Ansatz (außer dem von Grandy), der die Sicherheit einer Implementierung per Refinement sicherstellt, d.h. ein auf abstrakter Ebene spezifiziertes (und bewiesenes) Protokoll betrachtet und dieses per Verfeinerung als korrekt implementiert nachweist. Der dritte Ansatz, um die Korrektheit und Sicherheit von generiertem Quellcode zu garantieren, ist die Verifikation von Modelltransformationen, die den Code erzeugen. Auch hier gibt es bereits verschiedene Ansätze (z.B. [26, 187]), um Aussagen über konkrete Modelltransformationen

formal nachzuweisen. Jedoch ist mir keine Arbeit bekannt, die ein generiertes abstraktes formales Modell in Relation zu dem ebenfalls generierten Quellcode setzt und versucht, Aussagen über die Sicherheit des generierten Quellcodes (für alle mit den Transformationen entwickelten Anwendungen) zu machen.

Der Vergleich mit konkreten verwandten Arbeiten, die sich mit der Generierung von Code beschäftigen (z.B. [140, 145, 156, 182]), erfolgte bereits in den Abschnitten 6.4 und 3.4.

Teil V.

Anwendung des Ansatzes in der Praxis

11 Entwicklung großer und komplexer Anwendungen

Zusammenfassung: Anwendungen, die sehr komplex sind und eine große Menge an Funktionalität bereitstellen, können nur schwer in einem Schritt entworfen und realisiert werden. Für ihre Entwicklung bietet sich ein inkrementelles Vorgehen an, in dem zunächst nur ein Teil der geplanten Funktionen umgesetzt wird. In späteren Iterationen kann dieser Kern dann um weitere Protokolle ergänzt werden. Der SecureMDD-Ansatz unterstützt die inkrementelle Entwicklung großer Anwendungen. Das Endprodukt jeder Iteration ist ein lauffähiges, getestetes und verifiziertes (Teil-)System. Die Verifikation wird am Ende jeder Iteration für die zu diesem Zeitpunkt bestehende Teilanwendung durchgeführt. Werden hierbei bestimmte Regeln beachtet, ist die schrittweise Verifikation einer Anwendung ohne großen Mehraufwand möglich. Die Praxistauglichkeit des Ansatzes wurde anhand einer großen und komplexen Fallstudie, der „Elektronischen Gesundheitskarte“, evaluiert. Dieses Kapitel erläutert das inkrementelle Vorgehen im SecureMDD-Ansatz und beschreibt die Fallstudie. Beides ist auch in [131] publiziert.

Abschnitt 11.1 beschreibt das Vorgehen bei der inkrementellen Entwicklung einer Anwendung, Abschnitt 11.2 gibt einen Überblick über die Fallstudie „Elektronische Gesundheitskarte“ und Abschnitt 11.3 nennt einige Zahlen und Statistiken zu der Größe der durchgeführten Fallstudien und dem Aufwand für die Verifikation. Abschließend setzt Abschnitt 11.4 die in diesem Kapitel beschriebenen Ergebnisse in den Kontext anderer Forschungsarbeiten.

11.1. Inkrementelle Entwicklung

Die Kopierkartenanwendung sowie die ebenfalls betrachteten Fallstudien „Elektronischer Reisepass“, „Geldkarte“, „Altersverifikation“ und „Mondex“ (siehe Abschnitt 3.3) sind relativ kleine Anwendungen mit überschaubarer Funktionalität und nur wenigen Protokollen. Sie wurden mit dem SecureMDD-Ansatz in einer Iteration entwickelt. Dies bedeutet, dass die Anwendung zunächst vollständig mit UML modelliert und anschließend der Code generiert sowie die Sicherheit der Anwendungen bewiesen wurde. Für kleine Anwendungen hat sich folgendes Vorgehen bewährt:

1. Der erste Schritt ist das Erstellen des plattformunabhängigen UML-Modells der Anwendung. Dabei werden zunächst die möglichen Benutzer(gruppen) und Komponenten festgelegt. Außerdem ist es notwendig, sich Gedanken über mögliche Angriffspunkte sowie die Fähigkeiten des Angreifers zu machen. Es ist deshalb sinnvoll, zunächst das Deploymentdiagramm der zu entwickelnden Anwendung zu entwerfen. Anschließend werden die Klassen- und Sequenzdiagramme parallel erstellt, da die Diagramme voneinander abhängen. Sind die Kommunikationsschritte der Protokolle durch die Sequenzdiagramme beschrieben, müssen sie durch Aktivitätsdiagramme verfeinert werden. Hierfür ist es in den allermeisten Fällen notwendig, das zuvor erstellte Klassendiagramm um zusätzliche Datenklassen sowie Attribute und Assoziationen zu ergänzen. Neben den bisher beschriebenen, für die Transformationen direkt benötigten, UML-Diagramme, sind bei der Entwicklung einer Smart Card-Anwendung weitere Artefakte sinnvoll. Zum Beispiel ist es, je nach Anwendung, hilfreich, Use Case-Diagramme und -Beschreibungen zu erstellen. Diese können über die Beschreibung der Funktionalität hinaus auch eine Erläuterung möglicher Schwachstellen bzw. Angriffspunkte enthalten. Außerdem ist das Erstellen von Angriffsbäumen (Attack Trees) gut geeignet, um mögliche Schwachstellen der Anwendung systematisch zu erfassen. Bei diesem von Schneier in [172] vorgeschlagenen und von Saini et al. in [165] erweiterten Vorgehen wird in einer Baumstruktur aufgeschrieben, an welchen Stellen eine Anwendung potenziell angegriffen werden könnte. Die Use Cases und Angriffsbäume werden zurzeit bei der Generierung des Codes und formalen Modells von den Transformationen nicht verwendet.
2. Wenn das UML-Modell fertig ist, muss es validiert werden. In diesem automatisch ablaufenden Schritt wird überprüft, ob die in den Abschnitten 4.2 und 4.4 vorgestellten Modellierungsrichtlinien eingehalten werden. Außerdem wird geprüft, ob die in den Aktivitätsdiagrammen enthaltenen MEL-Ausdrücke syntaktisch korrekt und konsistent mit dem UML-Klassendiagramm sind, d.h. dass z.B. nur Objekte von einer Klasse erzeugt werden können, die im Klassendiagramm oder in den Sicherheitsdatentypen definiert ist. Wird bei der Validierung festgestellt, dass das Modell fehlerhaft ist oder gegen die Richtlinien verstößt, wird eine gut verständliche Fehlermeldung ausgegeben. Das UML-Modell muss dann entsprechend korrigiert werden. Ist die Validierung erfolgreich, kann das UML-Modell in Code und die formale Spezifikation transformiert werden.
3. Im Anschluss an die Validierung wird der Code der Anwendung (durch Ausführen der Modelltransformationen) generiert. Dieser ist bei erfolgreicher Validierung frei von Syntaxfehlern, muss aber auf funktionale Fehler und Sicherheitslücken getestet werden.
4. Um funktionale Fehler und Sicherheitslücken in dem generierten Code zu finden, kann das in Schritt 1 erstellte UML-Modell um Testfälle ergänzt werden (siehe Kapitel 7). Aus diesem erweiterten UML-Modell wird in einer weiteren Transformation Testfallcode generiert, der den generierten Anwendungscode testet. Stellt sich dabei heraus, dass der Code Protokollfehler oder Sicherheitslücken enthält, muss das UML-Modell entsprechend korrigiert werden. In diesem Fall sind die Schritte 2 bis 4 erneut auszuführen.
5. Ist der generierte Code frei von Fehlern und Sicherheitslücken folgt die Verifikation von Sicherheitseigenschaften für die modellierte Anwendung. Dieser Schritt ist optional und kann theoretisch weggelassen werden. In diesem Fall können jedoch keine Garantien bezüglich der Sicherheit der zu entwickelnden Anwendung gegeben werden. Für die Verifikation wird das formale Modell der Anwendung erzeugt und anschließend automatisch

in das Beweissystem KIV geladen. Dann werden die Sicherheitseigenschaften formuliert und verifiziert (siehe Kapitel 9). Stellt man bei der Verifikation fest, dass die Anwendung fehlerhaft bzw. unsicher ist, müssen die fehlerhaften Protokolle im UML-Modell korrigiert werden. In diesem Fall sind die Schritte 2 bis 5 nach der Korrektur erneut durchzuführen. Durch das Korrektheitsmanagement des Beweissystems können die bereits getätigten Beweise für die Protokollschritte wiederverwendet werden, die nicht von den gemachten Änderungen betroffen sind. Die Beweisschritte, die sich auf die geänderten Protokollschritte beziehen, werden jedoch ungültig und müssen wiederholt bzw. korrigiert werden.

Für kleinere Anwendungen mit wenigen Protokollen wird somit die Entwicklung der Anwendung in einer Iteration empfohlen. Kleinere Korrekturen an den Protokollen können dabei ohne großen Mehraufwand vorgenommen werden.

Betrachtet man jedoch große Fallstudien mit umfangreicher Funktionalität, ist die Entwicklung in einem Schritt nur schwer machbar. Ist die Anwendung sehr groß oder sollen zu einem späteren Zeitpunkt Funktionen ergänzt werden, ist ein inkrementelles Vorgehen unerlässlich. Die Idee hierbei ist es, nach jeder Iteration eine lauffähige, getestete sowie sichere Anwendung in den Händen zu halten.

Die inkrementelle Entwicklung einer Anwendung ist mit dem SecureMDD-Ansatz gut durchführbar. Innerhalb einer Iteration wird dabei ein Teil der Funktionalität der Anwendung, d.h. einige der kryptographischen Protokolle, entworfen und realisiert. Es bietet sich an, Gruppen von Funktionen zu bilden und diese zusammengehörigen Protokolle in einem Schritt zu realisieren. Zum Beispiel könnte man das Erstellen der Stamm- und Notfalldaten in der Gesundheitskartenanwendung in einem Schritt und das Erstellen und Einlösen eines Rezepts in der nächsten Iteration realisieren. Für jede Iteration ist das Vorgehen wie folgt:

1. Der Teil der Anwendung, der in der aktuellen Iteration realisiert werden soll, wird in UML modelliert. Dabei wird das bereits in den vorherigen Iterationen erstellte UML-Modell erweitert. Insbesondere werden die Klassendiagramme um zusätzliche Daten- und Nachrichtenklassen ergänzt und die Komponentenklassen erhalten weitere Attribute und Assoziationen. Da das Klassendiagramm für große Anwendungen schnell unübersichtlich wird, ist es besser, die Klassen auf mehrere Diagramme aufzuteilen. Sinnvoll ist es, ein Klassendiagramm für alle Komponenten- und Datenklassen sowie je ein weiteres Klassendiagramm für die Nachrichtenklassen jedes modellierten Protokolls zu definieren. Zusätzlich wird jedes Protokoll, das in der aktuellen Iteration entworfen wird, in einem eigenen Aktivitätsdiagramm modelliert. Das Ergebnis ist ein UML-Anwendungsmodell, das sowohl die Funktionalität der vorherigen als auch der aktuellen Iteration enthält.
2. Das UML-Modell wird validiert, d.h. das Modell wird auf Konsistenz und Syntaxfehler überprüft. Werden die Transformationen anschließend auf dem erfolgreich validierten Modell durchgeführt, ist das Ergebnis frei von Syntaxfehlern. Da die Validierung nicht viel Zeit kostet, wird jeweils das vollständige UML-Modell validiert.
3. Nach der Validierung wird das gesamte UML-Modell in lauffähigen Quellcode transformiert. Der in der vorherigen Iteration generierte Code wird dabei überschrieben, d.h. der gesamte Code wird neu generiert. Da die Generierung nur wenig Zeit in Anspruch nimmt, ist dies jedoch kein Problem.

4. Der im vorherigen Schritt generierte Code wird nun getestet. Dafür müssen für die Funktionen, die in dieser Iteration neu hinzugekommen sind, ebenfalls Testfälle modelliert werden. Anschließend wird für alle Tests (alte sowie neue) der Testfallcode generiert und der im vorherigen Schritt generierte Anwendungscode getestet. Schlägt ein Test fehl, müssen die fehlerhaften Protokolle im UML-Anwendungsmodell korrigiert und die Schritte 2 bis 4 anschließend erneut ausgeführt werden.
5. Der letzte Schritt dieser Iteration ist die Verifikation von Sicherheitseigenschaften für die hinzugekommenen Funktionen. Die Verifikation in einem inkrementellen Vorgehen wird im Folgenden im Detail beschrieben.

Ein inkrementelles Vorgehen birgt zwei potenzielle Probleme. Das erste Problem ist, dass einige der bewiesenen (Hilfs-)Eigenschaften möglicherweise nicht mehr gelten, wenn einer (Teil-)Anwendung zu einem späteren Zeitpunkt Funktionen, d.h. kryptographische Protokolle, hinzugefügt werden. So könnte zum Beispiel die Sicherheit eines Protokolls darauf beruhen, dass eine bestimmte Nonce geheim bleibt, d.h. dem Angreifer nicht bekannt ist. Ein anderes Protokoll könnte genau diese Nonce jedoch offenlegen, d.h. im Klartext versenden. Ein anderes Beispiel stammt aus der in Abschnitt 11.2 vorgestellten Fallstudie „Elektronische Gesundheitskarte“. Hier wurde bei der Verifikation eine Eigenschaft formuliert, die besagt, dass nur vertrauenswürdige Personen (dies sind in diesem Fall alle Ärzte und Apotheker, aber nicht die Patienten) die in einem elektronischen Rezept gespeicherte Nonce ansehen dürfen. Die Patienten bekommen zwar das Medikament ausgehändigt, können die Nonce jedoch nicht sehen. In einer späteren Iteration wurde dann die Funktion hinzugefügt, dass Patienten sich über ein Kiosksystem die Daten, die auf ihrer Karte gespeichert sind, anschauen können. Somit war die zuvor bewiesene Aussage nicht mehr gültig. Werden in einer späteren Iteration Protokolle hinzugefügt, müssen die bereits in den vorherigen Iterationen durchgeführten Invariantenbeweise für die neu hinzugekommenen Protokollschritte nachgeholt werden. Gilt eine Eigenschaft nach Hinzufügen weiterer Protokolle nicht mehr, wird dies an dieser Stelle bemerkt. Um das Problem zu umgehen, sollte die Formulierung und Verwendung von Hilfseigenschaften vermieden werden, die in einer späteren Iteration ungültig werden könnten. Stattdessen sollte die Verifikation auf allgemeingültigen Eigenschaften basieren, die voraussichtlich auch beim Hinzufügen weiterer Funktionen ihre Gültigkeit behalten.

Das zweite Problem ist, dass sich die Abstract State Machine beim Hinzufügen weiterer Protokollschritte ändert. Der Grund hierfür ist, dass für jeden Protokollschritt eine eigene ASM-Regel generiert wird. Dies hat zur Folge, dass alle Invariantenbeweise zunächst ungültig werden und erneut bewiesen werden müssen. Da die Verifikation interaktiv geschieht, bedeutet dies zusätzlichen Aufwand. Befolgt man jedoch einige Regeln, reduziert sich der zusätzliche Verifikationsaufwand in einem inkrementellen Vorgehen auf ein Minimum. Beim Hinzufügen von Funktionen müssen dann lediglich die in den vorherigen Iterationen durchgeführten Invariantenbeweise für die neu hinzugekommenen Protokollschritte nachgezogen werden.

Folgende drei Regeln sollten eingehalten werden, um den zusätzlichen Verifikationsaufwand, der durch ein inkrementelles Vorgehen entstehen kann, gering zu halten:

1. Es sollten niemals Änderungen an bestehenden und verifizierten Protokollen vorgenommen werden, sondern der Anwendung nur neue Protokolle hinzugefügt werden. Das Ändern eines bestehenden Protokolls bringt Änderungen an bestehenden Eigenschaften, Definitionen und bereits durchgeführten Beweisen mit sich, deren Korrektur viel Zeit kosten kann.

2. Die Invariantenbeweise sollten durch die Definition entsprechender Simplifikationsregeln (siehe Abschnitt 8.1) so optimiert werden, dass sie automatisch ablaufen. In diesem Fall bedeutet ein erneutes Ausführen der Beweise, dass sie für die Protokollschritte vorheriger Iterationen automatisch ablaufen.
3. Große Invarianten sollten in mehrere kleinere Invarianten aufgeteilt werden, so dass sich eine Invariante wenn möglich auf nur eine Komponente bezieht. Anstatt zum Beispiel eine Invariante zu verwenden, die besagt, dass der Angreifer keine Sitzungsschlüssel kennt, sollte es mehrere Invarianten der Form „Der Angreifer kennt den Sessionkey von Komponente A nicht“, „Der Angreifer kennt den Sessionkey von Komponente B nicht“ usw. geben. Kommt dann in einer späteren Iteration eine weitere Komponente hinzu, wird nur eine zusätzliche kleine Invariante benötigt und die Beweise für die bisherigen Invarianten behalten ihre Gültigkeit. Anderenfalls würden diese beim Hinzufügen einer weiteren Komponente ungültig werden

Werden diese drei Regeln befolgt, können die in den vorherigen Iterationen durchgeführten Beweise für die alten Protokollschritte automatisch wiederholt werden. Für die neuen Protokollschritte sind jedoch Interaktionen notwendig. Für diese werden Simplifikationsregeln hinzugefügt und bewiesen, so dass auch die neuen Protokollschritte in Zukunft automatisch bewiesen werden können. Sollte die Invariante nicht mehr gültig sein, muss eine neue kleine Invariante formuliert und bewiesen werden. Anschließend sollte der Invariantenbeweis automatisch durchführbar sein.

Zusammenfassend lässt sich sagen, dass das inkrementelle Durchführen von Beweisen im SecureMDD-Ansatz in etwa den gleichen Aufwand erfordert wie bei einem nicht-inkrementellen Vorgehen. Da das Wiederausführen der Beweise nach Hinzufügen von Funktionalität automatisch abläuft, ist der zusätzliche Aufwand hierfür vernachlässigbar. Ein inkrementelles Vorgehen hat jedoch den großen Vorteil, dass die formale Spezifikation und die Beweise für die Person, die die Verifikation durchführt, besser verständlich und übersichtlicher sind, da die Anwendung und damit auch die Größe der Spezifikation, erst von Iteration zu Iteration wächst. Es findet somit ein langsames Heranführen an die gesamte Anwendung statt, was bei der Verifikation sehr hilfreich ist.

11.2. Fallstudie „Elektronische Gesundheitskarte“

Die elektronische Gesundheitskarte wird zurzeit in Deutschland eingeführt und ersetzt die zuvor verwendete Versichertenkarte. Die Anwendung besteht nicht nur aus den Gesundheitskarten, sondern aus mehreren weiteren Komponenten wie zum Beispiel Heilberufsausweisen, eine Telematikinfrastruktur, Services sowie verschiedene Arten von Terminals. Die Einführung der Gesundheitskarte war ursprünglich schon für 2006 geplant, diese hatte sich aber aus verschiedenen Gründen immer wieder verzögert. Die jetzige Gesundheitskarte hat im Prinzip dieselbe Funktionalität wie die bisherige Versichertenkarte, d.h. sie dient der Speicherung der Versichertenstammdaten. Lediglich ein aufgedrucktes Foto des Versicherten ist neu hinzugekommen. Die Gesundheitskarte bildet jedoch die Basis für eine Modernisierung des Gesundheitssystems, bei der die Digitalisierung von medizinischen Daten (wie z.B. Notfalldaten und medizinische Daten wie Röntgenbildern, Arztbriefen und Diagnosen) eine zentrale Rolle spielt. Dafür soll die Anwendung der Gesundheitskarte schrittweise erweitert werden, z.B. um die elektronische Patientenakte und einen Organspendeausweis. Da die von der gema-

tik[1] vorgeschlagenen kryptographischen Protokolle für die Anwendung zurzeit obsolet sind, wurden für diese Fallstudie eigene Protokolle entworfen. Die von der gematik festgelegten Sicherheitsmerkmale wurden jedoch beim Entwurf berücksichtigt. Dies sind zum Beispiel die Verwendung von Zertifikaten für die Heilberufsausweise der Ärzte und Apotheker sowie die Gesundheitskarten und die Eingabe von PIN-Nummern durch die Patienten, Ärzte und Apotheker. Außerdem festgelegt ist, dass ein elektronisches Rezept von dem Heilberufsausweises des Arztes, der das Rezept ausstellt, signiert werden muss. Weiterhin kann ein Patient die Daten auf seiner Gesundheitskarte nur an einem extra dafür vorgesehenen Kiosk, der in einer Apotheke steht, einsehen.

Die für diese Fallstudie erstellten Diagramme sind zu groß, um sie vollständig in der Arbeit abzudrucken. Deshalb werden in diesem Abschnitt nur Ausschnitte der Modellierung gezeigt. Die Modelle können jedoch auf der Projektwebseite[2] eingesehen werden. Dort sind auch der generierte Anwendungscode sowie die durchgeführten Beweise zu dieser Fallstudie zu finden.

Das in Abbildung 11.1 dargestellte Deploymentdiagramm zeigt die an der Anwendung beteiligten Benutzer und Komponenten.

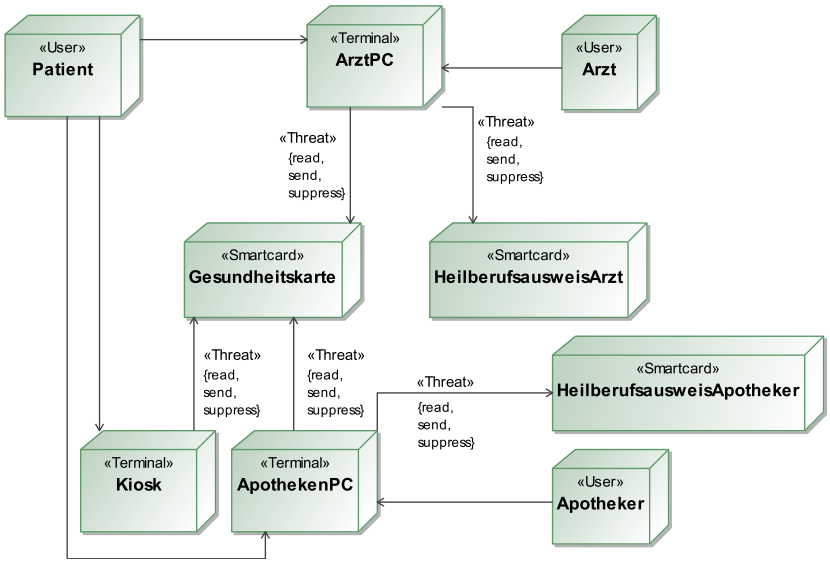

Abbildung 11.1.: Deploymentdiagramm der Gesundheitskarte

Jeder `Arzt` besitzt einen Heilberufsausweis (`HeilberufsausweisArzt`) und hat einen `ArztPC` in seiner Praxis stehen. Jeder `Patient` besitzt eine `Gesundheitskarte`. Geht ein Patient zum Arzt und möchte dieser eine Funktion der Gesundheitskartenanwendung nutzen, steckt er sowohl seinen Heilberufsausweis als auch die Gesundheitskarte des Patienten in zwei an den `ArztPC` angeschlossene Kartenleser.

Weiterhin sind die `Apotheker` an der Anwendung beteiligt. Jeder Apotheker besitzt ebenfalls einen Heilberufsausweis (`HeilberufsausweisApotheker`) und in jeder Apotheke steht ein `ApothekenPC`, der mit zwei Lesegeräten ausgestattet ist. Eines dieser Lesegeräte ist für die

[1] Gesellschaft für Telematikanwendungen der Gesundheitskarte mbH, www.gematik.de
[2] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/

Gesundheitskarte, der andere für den Heilberufsausweis vorgesehen. Des Weiteren steht in einigen Apotheken ein Terminal, `Kiosk` genannt, an dem jeder Patient die Daten, die auf seiner Gesundheitskarte gespeichert sind, einsehen kann. An jedem Kiosk ist ein Lesegerät für eine Gesundheitskarte vorgesehen. Für die Anwendung wird angenommen, dass ein Angreifer vollen Zugriff (d.h. lesen, schreiben und Nachrichten unterdrücken) auf alle Kommunikationskanälen zwischen einer Smart Card und einem Terminal hat. Das Problem, dass ein Angreifer dem Patienten oder Arzt im Behandlungsraum der Arztpraxis oder beim Einlösen eines Rezepts in der Apotheke über die Schulter schaut und auf diese Weise geheimen Daten mitliest, ist separat von der Sicherheit der kryptographischen Protokolle der Anwendung zu betrachten. Die Kommunikationskanäle zwischen den Benutzern der Anwendung und den Terminals sind deshalb nicht mit dem Stereotyp ≪Threat≫ annotiert.

Für die Gesundheitskartenanwendung wurden fünfzehn Klassendiagramme erstellt. Eines davon enthält die Komponentenklassen sowie die von den Komponenten verwendeten Datenklassen. Die weiteren Diagramme enthalten die Nachrichten- und assoziierten Datenklassen der vierzehn entworfenen Protokolle, d.h. für die Nachrichtenklassen jedes Protokolls wurde ein eigenes Diagramm verwendet. Dies dient einer besseren Strukturierung des Projekts. Insgesamt besteht die Gesundheitskartenanwendung aus 151 Klassen.

Abbildung 11.2 zeigt einen Ausschnitt aus einem Klassendiagramm der Anwendung. Der Ausschnitt zeigt die Smart Card-Komponente `Gesundheitskarte` sowie einige Datenklassen. Die Gesundheitskarte hat z.B. ein Attribut `pin`, in dem die PIN-Nummer des Patienten gespeichert ist sowie ein Attribut `privkeyEGK`, das den privaten Schlüssel der Gesundheitskarte speichert. Zu jeder Gesundheitskarte gehört ein Zertifikat, das durch die Klasse `ZertifikatPatient` repräsentiert wird. Über die Attribute der Klasse `ZertifikatPatientData` sind die zu dem Zertifikat gehörenden Daten modelliert. Dies sind der Name `name` des Patienten, der zu dem im Attribut `privkeyEGK` gespeicherte öffentliche Schlüssel `pubkey`, die Versichertennummer des Patienten (`versichertennr`) und eine eindeutige Kennzahl für die Krankenkasse des Patienten (`krankenkassenID`). Außerdem enthält das Zertifikat eine `id`, die angibt, dass es sich bei dem Zertifikat um eines für eine Gesundheitskarte handelt. Dieses Attribut ermöglicht einem Kommunikationspartner bei der Authentisierung festzustellen, mit welchem Komponententyp (Heilberufsausweis, Gesundheitskarte oder Kiosk) zurzeit kommuniziert wird. Die Gesundheitskarte speichert außerdem die Notfalldaten des Patienten (über das Assoziationsende `notfalldaten`). Die eigentlichen Daten sind als String modelliert. Die Klasse `Notfalldaten` besitzt außerdem eine Nonce `nonce`, die beim Erstellen der Notfalldaten benötigt wird. Auf der Karte gespeichert sind außerdem die persönlichen Daten des Versicherten (`Versichertenstammdaten`) wie z.B. das Ablaufdatum der Karte oder die Kennnummer der Krankenkasse. Eine Gesundheitskarte kann bis zu fünf elektronische Rezepte speichern. Dies ist über die Assoziation zu der Klasse `ERezept` modelliert. Ein elektronisches Rezept besteht aus drei Teilen: den eigentlichen Rezeptdaten (Assoziationsende `rezeptdaten`), der Signatur `sig` über den Rezeptdaten (die von dem Arzt, der das Rezept ausstellt, erstellt wird) sowie dem Zertifikat des Arztes, der das Rezept ausgestellt hat (`zertifikatAusstellenderArzt`). Das Zertifikat wird beim Einlösen des Rezeptes verwendet, um die Signatur zu verifizieren. Die Rezeptdaten, modelliert durch die Klasse `ERezeptdaten`, bestehen aus einer Nonce, die für ein Rezept eindeutig ist, dem Ausstellungsdatum sowie der Einnahmeanweisung. Außerdem sind in den Rezeptdaten die `Versichertenstammdaten` des Patienten, für den das Rezept ausgestellt wurde sowie

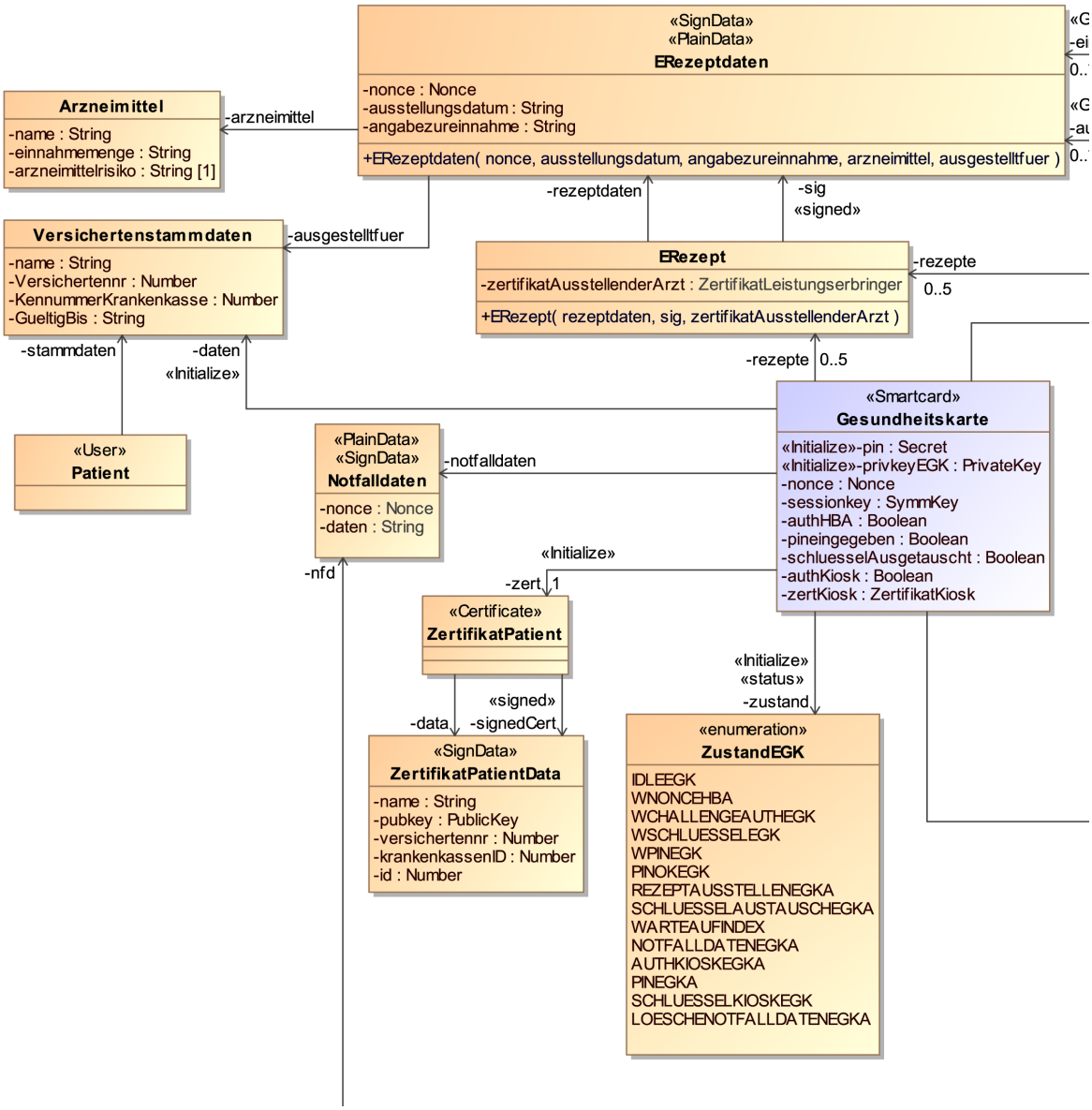

Abbildung 11.2.: Ausschnitt aus einem Klassendiagramm der Gesundheitskartenanwendung

das verschriebene `Arzneimittel` gespeichert. In der Klasse `ZustandEGK` sind zudem alle Zustände definiert, in denen sich eine Gesundheitskarte befinden kann.

Die für die Anwendung entworfenen Protokolle lassen sich in Basisprotokolle sowie Protokolle, die die eigentliche Funktionalität realisieren, unterteilen. Für die Gesundheitskartenanwendung wurden vierzehn Protokolle entworfen. Sieben davon sind Basisprotokolle, die weiteren sieben realisieren eine bestimmte Funktionalität der Anwendung.

Folgende Basisprotokolle wurden für die Anwendung entworfen:

- Beidseitige Authentisierung zwischen einem Heilberufsausweis und einer Gesundheitskarte
- Beidseitige Authentisierung zwischen einer Gesundheitskarte und einem Kiosk
- Austausch eines Sitzungsschlüssels zwischen einem Terminal (ArztPC oder ApothekenPC), einer Gesundheitskarte und einem Heilberufsausweis
- Austausch eines Sitzungsschlüssels zwischen einer Gesundheitskarte und einem Kiosk
- Eingabe und Überprüfung der PIN-Nummer des Patienten am ArztPC bzw. ApothekenPC
- Eingabe und Überprüfung der PIN-Nummer des Patienten am Kiosk
- Eingabe und Überprüfung der PIN-Nummer des Arztes am ArztPC bzw. des Apothekers am ApothekenPC

Die weiteren entworfenen Protokolle sind:

- Anlegen von Notfalldaten durch einen Arzt
- Ändern der Notfalldaten durch einen Arzt
- Löschen der Notfalldaten durch den Patienten am Kiosk
- Erstellen eines Rezepts an einem ArztPC und Speicherung des Rezepts auf einer Gesundheitskarte
- Einlösen eines auf der Gesundheitskarte gespeicherten Rezepts in einer Apotheke
- Anzeige der auf einer Gesundheitskarte gespeicherten Rezepte an einem Kiosk
- Anzeige der auf der Gesundheitskarte gespeicherten Versichertenstammdaten sowie der Notfalldaten des Patienten am Kiosk

Die Basisprotokolle Authentisierung, Austausch eines Sitzungsschlüssels sowie Eingabe und Überprüfung der PIN-Nummer(n) werden immer vor den Protokollen ausgeführt, die die Funktionen der Anwendung realisieren. Die Hintereinanderausführung der Protokolle ist dabei durch boolesche Attribute sichergestellt. Zum Beispiel speichert eine Gesundheitskarte in dem Attribut `authHBA`, ob die Authentisierung des Heilberufsausweises (des Arztes oder Apothekers) erfolgreich war. Wird das Protokoll für die Eingabe der PIN-Nummer ausgeführt, prüft die Karte zunächst, ob der Wert des Attributs auf `true` gesetzt ist und bricht anderenfalls den Protokolllauf ab.

Die Anwendung wurde in drei Iterationen entworfen. Zunächst wurden die Basisprotokolle, an denen die Arzt- und ApothekenPCs beteiligt sind, realisiert. In der zweiten Iteration wurden die Protokolle für das Erstellen und Einlösen eines Rezepts sowie das Anlegen und Ändern der Notfalldaten entworfen und realisiert. In der dritten Iteration folgten die Basisprotokolle, an denen ein Kiosk beteiligt ist, sowie die Funktionen „Löschen der Notfalldaten“ und „Anzeige der Stammdaten sowie der auf einer Gesundheitskarte gespeicherten Rezepte“. In Tabelle 11.1 werden die durchgeführten Iterationen in Beziehung zu den Komponententypen, die an den in der Iteration entworfenen Protokollen beteiligt sind, gesetzt.

Für die Gesundheitskartenanwendung wurden verschiedene Sicherheitseigenschaften bewiesen. Eine Eigenschaft besagt, dass alle Rezepte, die in einer Apotheke eingelöst werden, zuvor auf einem ArztPC erstellt wurden. Da angenommen wird, dass die Terminals frei von Schadsoftware sind und der Angreifer somit keinen Zugriff auf diese Geräte hat, impliziert dies,

Iteration	Gesundh.-karte	HBA Arzt	HBA Apotheker	Arzt-PC	Apot.-PC	Kiosk
1	X	X	X	X	X	
2	X	X	X	X	X	
3	X					X

Tabelle 11.1.: Angabe, welche Komponenten von den Erweiterungen der jeweiligen Iteration betroffen sind

dass in den Apotheken nur echte Rezepte einlösbar sind. Das heißt es ist nicht möglich, Rezepte zu fälschen. Eine zweite verifizierte Eigenschaft ist, dass ein Rezept nur einmal eingelöst werden kann. Jedes Rezept enthält eine eindeutige Nonce (siehe Klasse `ERezeptdaten` in Abbildung 11.2). Die Eigenschaft besagt, dass eine Liste bestehend aus allen Rezepten, die in allen Apotheken eingelöst wurde, keine Rezepte mit gleicher Nonce enthält. Dies impliziert, dass es nicht möglich ist, ein Rezept zu kopieren und sowohl das Original als auch die Kopie einzulösen, um sich die doppelte Menge eines Medikaments ausgeben zu lassen. Eine weitere Eigenschaft ist, dass die Rezepte vor dritten Personen geheim bleiben, d.h. der Angreifer erfährt diese Daten nicht.

Wie in Kapitel 9 erläutert, werden für den Nachweis der Sicherheit Invariantenbeweise verwendet, d.h. es wird gezeigt, dass die Sicherheitseigenschaft invariant über einem (beliebigen) ASM-Schritt ist. In der Regel ist die zu beweisende Sicherheitseigenschaft für sich genommen jedoch nicht invariant über einem Schritt, sondern es werden weitere (Hilfs-)Eigenschaften für den Beweis benötigt, die ebenfalls als invariant nachgewiesen werden müssen. In Abbildung 11.3 ist der Ausschnitt aus dem Spezifikationsgraph der Anwendung dargestellt, der sich überhalb der ASM-Spezifikation befindet und in dessen Spezifikationen die für die Verifikation der Gesundheitskarte benötigten Hilfseigenschaften definiert sind.

Jede Inv..-Spezifikation (in der Grafik) enthält eine oder mehrere (Hilfs-)Invarianten, die von den darunterliegenden Spezifikationen abhängig sind. Um zu zeigen, dass nur echte Rezepte eingelöst werden können (`Inv-Rezepte-echt`), benötigt man die Eigenschaft, dass der Angreifer niemals einen symmetrischen Sitzungsschlüssel erfährt (`Inv-no-SymmKeys`). Der Austausch eines Sitzungsschlüssels erfolgt durch Verschlüsselung dieses Schlüssels mit dem öffentlichen Schlüssel einer Komponente. Diese Eigenschaft setzt deshalb voraus, dass der Angreifer einen eigenen öffentlichen Schlüssel nicht in die Anwendung einschmuggeln, d.h. den beteiligten Komponenten nicht als „vertrauenswürdigen öffentlichen Schlüssel“ aufdrängen kann (`Inv-no-PrivateKeys`). Außerdem ist der Austausch eines Sitzungsschlüssels durch Verwendung einer Nonce in einem Challenge-Response-Verfahren gegen Replay-Angriffe geschützt. Dies setzt voraus, dass der Angreifer niemals eine Nonce kennt, die aktuell in einem Protokolllauf als Challenge verwendet wird. Darüber hinaus werden weitere Hilfseigenschaften benötigt, die alle als invariant nachgewiesen werden mussten.

Während der Entwicklung wurden zwei Fehler in den Protokollen der Anwendung entdeckt. Der erste war eine fehlende Nonce, so dass das Wiedereinspielen einer Nachricht (Replay) möglich war. Der zweite Fehler betraf die Heilberufsausweise der Ärzte und Apotheker. Um den Heilberufsausweis eines Apothekers von dem eines Arztes unterscheiden zu können, ist in den Zertifikaten, die auf den Ausweisen gespeichert sind, in einem booleschen Attribut gespeichert welchen Typ der Heilberufsausweis hat (d.h. `HeilberufsausweisArzt` oder `HeilberufsausweisApotheker`). Dieses Attribut ist analog zu dem Attribut `id` des Zer-

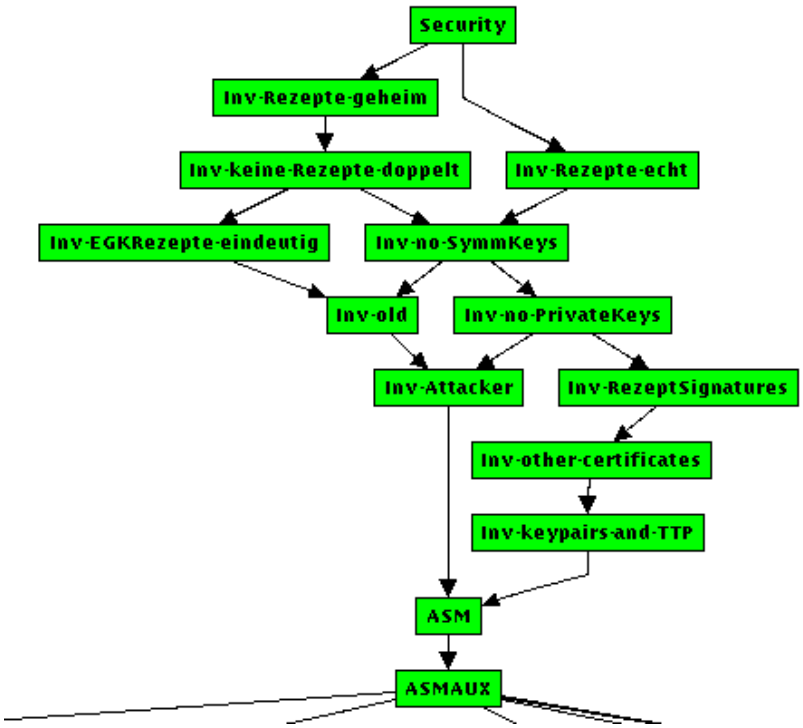

Abbildung 11.3.: Teil des Spezifikationsgraphs der Gesundheitskartenanwendung

tifikats einer Gesundheitskarte (siehe Abbildung 11.2) definiert. In einigen Protokollen muss überprüft werden, um welchen Kartentyp es sich handelt. Zum Beispiel muss beim Erstellen eines Rezepts überprüft werden, dass hierfür der Heilberufsausweis eines Arztes verwendet wird. In einigen Protokollen fehlte diese Überprüfung.

11.3. Statistiken

Dieser Abschnitt soll einen Überblick über die Größe sowie den Verifikationsaufwand der beiden betrachteten Fallstudien „Kopierkarte“ und „Elektronische Gesundheitskarte“ geben.

Die Kopierkartenanwendung besteht aus einem Klassendiagramm mit 25 Klassen. Die drei realisierten Protokolle bestehen aus insgesamt 17 Protokollschritten und sind in jeweils einem Sequenz- und einem Aktivitätsdiagramm modelliert. Außerdem enthält das UML-Anwendungsmodell das Deploymentdiagramm der Anwendung.

Für die Kopierkartenanwendung wurden 7346 Zeilen Anwendungscode generiert (inklusive der Kommentare). Dieser lässt sich aufteilen in den Java Card-Code, der auf die Kopierkarten geladen wird (2052 Zeilen), den Code für die Kopiergeräte (2644 Zeilen) und den Code für die Ladegeräte (2650 Zeilen). Da einige Klassen, z.B. die Sicherheitsdatentypen, von mehreren Komponenten verwendet werden und für jede Komponente ein eigenes Java-Package mit dem Quellcode der Komponente generiert wird, sind jedoch Teile des generierten Codes redundant. Betrachtet man den generierten Java Card-Code im Detail, sieht man, dass die Klasse

`Coding` mit 577 Zeilen am größten ist. Diese enthält die (De-)Serialisierung der Nachrichten- und Datenobjekte. Am zweitgrößten ist mit 286 Zeilen der Objektmanager, gefolgt von der Klasse `SimpleComm` mit 198 Zeilen. Die Klasse `Copycard`, die die Implementierung der Protokollschritte für die Kopierkarte enthält, hat 164 Zeilen. Die Daten- und Nachrichtenklassen sind zwischen 30 und 60 Zeilen lang.

Für die Verifikation der Sicherheitseigenschaft „Es geht kein Geld verloren“ hat eine Person zwei Wochen benötigt. Dies umfasst sowohl die Zeit zum Finden und Korrigieren von Designfehlern in den Protokollen als auch das erneute Erstellen der Beweise nach Änderung einiger Protokollschritte. Eine Schwierigkeit bei der Verifikation war das Finden geeigneter Hilfsinvarianten. Außerdem war das Finden der Invariante für den Sicherheitsnachweis (d.h. das Betrachten der `Load`- und `Pay`-Nachrichten, die sich „in Transit“ befinden, siehe Abschnitt 9.2) schwierig und die Invariante war erst nach mehreren Anläufen korrekt formuliert. Für die Beweise wurden ungefähr 3.000 Benutzerinteraktionen, zum Beispiel die Instanziierung eines Quantors, benötigt. Davon waren ca. 2.500 Interaktionen für die Sicherheitsbeweise und 500 für das Beweisen der Hilfseigenschaften nötig. Dies ist ungefähr ein Drittel des Aufwandes, der von Haneberg für die Verifikation dieser Fallstudie benötigt wurde [79].

Die „Elektronische Gesundheitskarte“ besteht aus 15 Klassendiagrammen mit insgesamt 151 Klassen. Die vierzehn realisierten Protokolle der Anwendung haben insgesamt 96 Protokollschritte. Neben den Sequenz- und Aktivitätsdiagrammen für die Protokolle enthält das UML-Modell außerdem das Deploymentdiagramm. Der generierte Anwendungscode ist 82.000 Zeilen lang, davon fallen 36.000 Zeilen auf den Java Card-Code für die Smart Cards und 46.000 Zeilen auf den Java-Code für die Terminals. Wie auch bei der Kopierkartenanwendung sind einige Klassen redundant, da der Code für jede Komponente in ein eigenes Paket generiert wird. Für die Verifikation der Sicherheitseigenschaften hat eine Person einen Monat benötigt. Die Beweise bestehen aus ungefähr 60.000 Beweisschritten. Ein Beweisschritt ist zum Beispiel eine prädikatenlogische Simplifikation oder die Ausführung einer Zuweisung. Außerdem waren 4.000 Benutzerinteraktionen notwendig. Zusätzlich zu den Sicherheitseigenschaften mussten ungefähr 800 Hilfstheoreme bewiesen werden.

11.4. Verwandte Arbeiten

Der wohl bekannteste Prozess für die Entwicklung sicherer Systeme ist der Security Development Lifecycle (SDL) [87, 126] von Microsoft. Seit Windows Vista wurde jede von Microsoft entwickelte Software SDL-konform erstellt. Microsoft unterstützt auch andere Firmen bei der Einführung von SDL. So wird der Prozess unter anderem von Adobe, Google, Oracle und Symantec verwendet. SDL folgt dem „Security by Design“-Prinzip. Dies bedeutet, dass Sicherheit ein fester Bestandteil des gesamten Entwicklungsprozesses ist, durchgängig in diesen integriert ist und ganzheitliche Maßnahmen getroffen werden, um die Sicherheitsanforderungen zu berücksichtigen. Das Ziel von SDL ist, bestehende Entwicklungsprozesse um Sicherheitsmaßnahmen zu ergänzen. Diese lassen sich parallel zu den Entwicklungsschritten ausführen. SDL ist eine Anleitung, wie Anwendungen sicher entworfen und realisiert werden können. Diese sind allgemein gehalten und für die Entwicklung jeder Art von Software geeignet.

Andere Entwicklungsprozesse (z.B. die in [54] und [36] vorgestellten) verwenden ebenfalls das „Security by Design“-Prinzip und definieren Sicherheit als integralen Bestandteil des Prozesses. Diese Arbeiten stellen allgemeine Prozesse für die Entwicklung von Anwendungen

zur Verfügung, die nicht auf einen speziellen Problembereich zugeschnitten sind, wie z.B. die Entwicklung von Anwendungen, die auf kryptographischen Protokollen basieren.

In [98] beschäftigen sich Jürjens et al. mit der Sicherheit von langlebigen, sich über die Zeit ändernen Systemen. Ziel ist es, die über die Zeit gemachten Änderungen an der Software in den zugehörigen UML-Modellen zu erfassen und formal nachzuweisen, dass die neuen Softwareversionen ebenfalls sicher sind. Dies geschieht durch explizites Modellieren und Verifikation der Änderungen auf UML-Modellierungsebene. Hierfür erweitert Jürjens das UMLSec-Tool [89] um neue Stereotypen und Tags. Für die Verifikation werden automatische Beweistools verwendet. Mögliche Änderungen an einem System sind das Hinzufügen, Löschen und Austauschen von Modellelementen. Ein inkrementeller Entwicklungsprozess wird jedoch nicht vorgestellt.

Die Anwendung „Elektronische Gesundheitskarte“ wurde bereits von mehreren Gruppen betrachtet. In [6] diskutieren Apel et al. ein modellbasiertes Verfahren zur Testfallgenerierung und evaluieren dieses am Beispiel der Elektronischen Gesundheitskarte. Die Sicherheit der Anwendung wird jedoch nicht untersucht. Kardas und Tunali beschreiben in [103] den Entwurf und die Implementierung eines eigenen, Smart Card-basierten Gesundheitskartensystems. Auf eine inkrementelle Entwicklung wird jedoch nicht eingegangen. In [95] werden ebenfalls Sicherheitseigenschaften auf Modellebene für die deutsche Gesundheitskartenanwendung bewiesen. Es werden Standardsicherheitseigenschaften (wie Integrität und Vertraulichkeit) betrachtet und kein Quellcode generiert. Es sind jedoch keine Arbeiten bekannt, die anwendungsspezifische Sicherheitseigenschaften, wie zum Beispiel die Eigenschaft, dass Rezepte nicht doppelt eingelöst werden können, für die Gesundheitskartenanwendung nachgewiesen haben.

12 Toolunterstützung

Zusammenfassung: Der SecureMDD-Entwicklungsansatz ist vollständig werkzeugunterstützt. Nach Erstellung des UML-Anwendungsmodells lässt sich dieses mit drei bis vier Mausklicks exportieren und in Quellcode sowie in die formale Spezifikation transformieren. Auch das Einlesen des formalen Modells im Beweissystem funktioniert automatisch. Dieses Kapitel stellt die in SecureMDD verwendete Werkzeugkette vor und beschreibt den Weg von einem UML-Modell bis zum Code bzw. der Durchführung der Verifikation basierend auf dem generierten formalen Modell.

Für die Modellierung mit UML kann ein beliebiges UML-Modellierungstool verwendet werden, mit dem der Export eines Modells in das Eclipse-UML2-XMI-Format möglich ist. Bei den im Rahmen dieser Arbeit durchgeführten Fallstudien wurde mit dem Tool Magic Draw[1] gearbeitet.

Das exportierte UML-Modell kann dann in Eclipse[2] eingelesen und anschließend transformiert werden. Hierfür gibt es ein Eclipse-Plugin, das die Ausführung der Transformationen nach einem Rechtsklick auf das exportierte UML-Modell ermöglicht. Abbildung 12.1 zeigt die Entwicklungsumgebung Eclipse, in der nach einem Rechtsklick auf ein exportiertes UML-Modell die durchzuführende Transformation auswählbar ist.

Vor Durchführung der Generierung von Code oder der formalen Spezifikation muss das plattformunabhängige UML-Modell validiert werden. Bei der Validierung wird geprüft, ob das Modell ein gültiges SecureMDD-Modell ist, d.h. ob alle in Kapitel 4 beschriebenen Modellierungsrichtlinien eingehalten werden. Dieser Validierungsschritt wird automatisch vor Ausführung der Modelltransformationen durchgeführt, ist aber auch einzeln aufrufbar (*validate SecureMDD uml model*). Tritt bei der Validierung ein Fehler auf, d.h. verstößt das UML-Modell gegen die vorgegebenen Richtlinien und kann nicht fehlerfrei transformiert werden, wird der entsprechende Fehler in gut verständlicher Form über die Konsole ausgegeben.

Die Generierung des Quellcodes kann dann in einem Schritt erfolgen. In diesem Fall werden alle plattformspezifischen Modelle sowie der Quellcode für die Terminals und die Smart Cards durch Rechtsklick auf den Eintrag *convert pim to code* erzeugt. Die Transformationen können aber auch einzeln hintereinander ausgeführt werden.

[1] http://www.nomagic.com/products/magicdraw.html
[2] http://www.eclipse.org/modeling/

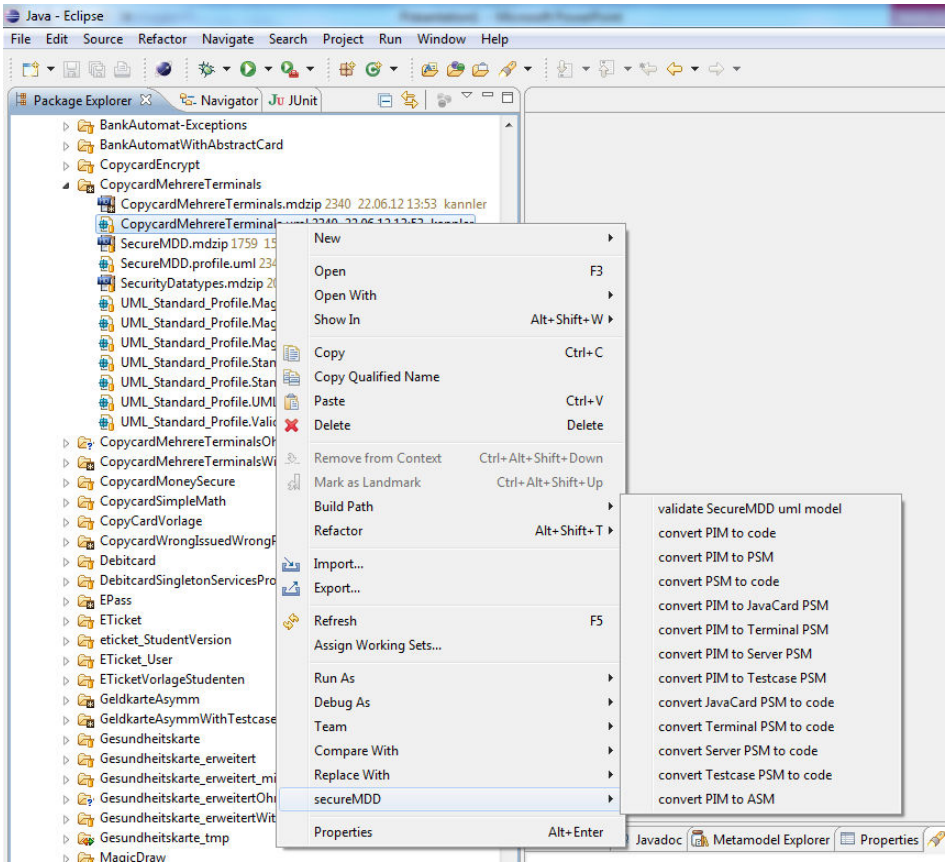

Abbildung 12.1.: Ausführen von Transformationen durch Rechtsklick auf das exportierte UML-Modell

Die Modell-zu-Modell-Transformationen sind in QVTO [149] implementiert. QVTO ist eine operationale Transformationssprache, von der es mehrere Implementierungen gibt. In dieser Arbeit wurde die Implementierung des Eclipse Modeling Projects (Teilprojekt M2M) verwendet. Die Modell-zu-Text-Transformationen sind in den Sprachen Xpand und Xtend implementiert. Beide sind Teil der oAW5-Plattform[3], die ebenfalls in das Eclipse Modeling Project (Teilprojekt M2T) integriert ist. Die Modelltransformationen sind in der Arbeit nicht abgedruckt, können aber in [129] und [128] nachgelesen werden.

Um den Testcode aus einem plattformunabhängigen UML-Modell, in dem Testfälle modelliert sind, zu generieren, ist ebenfalls lediglich ein Rechtsklick auf das UML-Modell notwendig. Aus technischen Gründen ist diese Transformation in zwei Schritte unterteilt: Die Generierung eines plattformspezifischen Testfallmodells sowie anschließend die Generierung des Codes aus diesem PSM. Enthält das UML-Modell keine Testfälle, wird eine Fehlermeldung ausgegeben. Wie auch bei der Transformation eines Anwendungsmodells wird vor der Generierung des Testfallcodes eine Validierung des UML-Modells durchgeführt um sicherzustellen, dass das UML-Modell den Modellierungsrichtlinien genügt. Hat man den Anwendungscode sowie den Testfallcode generiert, lässt sich der Testfallcode mit einem Rechtsklick auf die generierte

[3] www.openarchitectureware.org

Datei `AllTests.java` als JUnit Test[4] ausführen (*Run as .. JUnit Test*). Die Ergebnisse der durchgeführten Tests werden dann grafisch anhand eines (grünen oder roten) Ergebnisbalkens übersichtlich in Eclipse angezeigt. Damit der lästige Schritt des Ladens des Smart Card-Codes auf die Chipkarten beim Testen entfällt, wird eine Simulation für Smart Cards zur Verfügung gestellt, die sich (in den allermeisten Fällen) genauso verhält, als würde der Code auf einer echten Smart Card ausgeführt.

Die aus dem UML-Modell generierte formale Spezifikation lässt sich automatisch in den Theorembeweiser KIV[5] [9] einlesen. Werden bei der Verifikation Fehler im Anwendungsmodell entdeckt, können diese im UML-Modell korrigiert und das formale Modell anschließend neu generiert und in das Beweissystem geladen werden. Durch das Korrektheitsmanagement des Beweistools ist sichergestellt, dass die schon gemachten Beweise weiterhin gültig sind (sofern sie nicht von den Änderungen betroffen sind).

Für das Parsen der in den UML-Aktivitätsdiagrammen enthaltenen MEL-Ausdrücke wird ANTLR[6] verwendet. Das Erzeugen eines annotierten abstrakten Syntaxbaums ist als Eclipse-Plugin implementiert. Die Übersetzung der MEL-Ausdrücke in Quellcode bzw. ASM-Code ist in Java programmiert. Die entsprechenden Methoden hierfür werden von den QVT- und Xpand-Transformationen aus aufgerufen.

Zusammenfassend lässt sich sagen, dass der gesamte Ansatz vollständig von Tools unterstützt wird. Der Export des UML-Modells und das Ausführen der Transformationen sind mit nur wenigen Benutzerinteraktionen durchführbar und leicht zu bedienen.

[4] http://www.junit.org/
[5] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/
[6] www.antlr.org

13 Ergebnisse, Erfahrungen und Ausblick

Zusammenfassung: Das Hauptresultat dieser Dissertation ist ein modellgetriebener Ansatz für die Entwicklung sicherer Smart Card-Anwendungen. Der Ansatz integriert die formale Verifikation und ermöglicht es, Garantien für die Sicherheit der entwickelten Anwendungen zu geben. Desweiteren unterstützt der Ansatz die Generierung von lauffähigem Quellcode.

Abschnitt 13.1 fasst die Ergebnisse der Arbeit noch einmal zusammen. Abschnitt 13.2 gibt einen Ausblick auf mögliche weiterführende Forschungsarbeiten, die auf den hier vorgestellten Ergebnissen aufbauen.

13.1. Erreichte Ergebnisse und Erfahrungen

In dieser Arbeit wurde ein modellgetriebener Softwareentwicklungsansatz für die Entwicklung sicherer Smart Card-Anwendungen vorgestellt. Diese Anwendungen basieren auf kryptographischen Protokollen, die sehr anfällig für Fehler sind und deren Entwurf schwierig ist. In der Praxis kommt es immer wieder vor, dass Anwendungen erst nach jahrelangem Einsatz als unsicher und fehlerhaft erkannt werden (z.B. [61, 141]). Somit ist es für die Entwicklung sicherer Smart Card-Anwendungen essentiell, dass zum einen der Aspekt der Sicherheit der Anwendung vollständig in den Ansatz integriert ist. Zum anderen ist es erforderlich, Garantien bezüglich der Sicherheit der entwickelten Anwendungen geben zu können. Dies kann durch den formalen Nachweis der Sicherheit auf einem abstrakten Modell der Anwendung gelöst werden (z.B. [12, 34, 117, 152]). Damit die entwickelte Anwendung jedoch sicher ist, müssen die verifizierten Sicherheitseigenschaften nicht nur auf dem abstrakten Modell, sondern auch für die Implementierung gelten, die außerdem fehlerfrei sein muss. Nach bestem Wissen ist dies die erste Arbeit, die einen modellgetriebenen Entwicklungsansatz zur Verfügung stellt, der den formalen Nachweis von (anwendungsspezifischen) Sicherheitseigenschaften für Smart Card-Anwendungen unterstützt und gleichzeitig lauffähigen Code generiert, für den die bewiesenen Eigenschaften ebenfalls gelten.

Der SecureMDD-Ansatz ermöglicht die Modellierung einer Smart Card-Anwendung mit UML. Hierfür wurde die Unified Modeling Language durch die Definition eines UML-Profils, von Sicherheitsdatentypen sowie der domänenspezifischen Sprache MEL auf das Gebiet der sicherheitskritischen (Smart Card)-Anwendungen zugeschnitten. Die Modellierung ist leicht erlern-

bar und somit von Entwicklern, die sich bereits mit UML auskennen, einfach zu verwenden. Dies wurde in Lehrveranstaltungen, in denen Studenten Anwendungen mit dem SecureMDD-Framework entworfen haben, evaluiert. Außerdem abstrahiert die Modellierung stark von den technischen Details der Implementierungsebene sowie der formalen Spezifikation. Der Modellierer hat somit die Möglichkeit, sich auf den Entwurf der Protokolle zu konzentrieren, ohne sich um die technische Realisierung Gedanken machen zu müssen.

In dieser Arbeit wurde außerdem die vollautomatische Generierung von lauffähigem Quellcode aus dem UML-Anwendungsmodell realisiert. Die Modelltransformationen generieren sowohl auf Smart Cards lauffähigen Java Card-Code als auch Java-Code für die Terminals der Anwendung. Für die Smart Cards wird, anders als für Chipkarten üblich, objekt-orientierter Code generiert. Dieser ist sehr viel besser lesbar und verständlicher als „klassischer“, auf Byte-Arrays direkt operierender Smart Card-Code. Weiterhin werden bei der Generierung des Chipkartencodes der begrenzte Speicherplatz und die fehlende Garbage Collection auf Smart Cards berücksichtigt und als Lösung ein Objektmanager zur Objektverwaltung erstellt. Für die Byte-Array-basierte Kommunikation zwischen einem Terminal und einer Smart Card sowie die Durchführung der kryptographischen Operationen wird außerdem eine (De-)Serialisierungsschicht für Objekte generiert.

Der SecureMDD-Ansatz unterstützt das modellbasierte Testen einer Anwendung. Hierfür können Testfälle mit UML modelliert und aus dem Modell automatisch Code generiert werden, der den (ebenfalls generierten) Anwendungscode testet. Die Testunterstützung ermöglicht das einfache und leichte Finden sowohl von logischen Protokollfehlern als auch von Sicherheitslücken.

Ein weiteres wichtiges Ergebnis dieser Arbeit ist die automatische Generierung eines formalen Modells der modellierten Anwendung, um Sicherheitseigenschaften verifizieren zu können. Das formale Modell ist eine Nachbildung der realen Welt und basiert auf algebraischen Spezifikationen sowie Abstract State Machines. Es kann im interaktiven Theorembeweiser KIV [9] für den formalen Sicherheits- und Korrektheitsnachweis verwendet werden. Die Verifikation einer Eigenschaft geschieht durch die symbolische Ausführung von DL-Programmen sowie Invariantenbeweisen. Das formale Modell ist eine Erweiterung des Prosecco-Ansatzes. Mit dieser Arbeit konnte die dort vorgestellte Verifikation jedoch deutlich vereinfacht und verbessert werden. Eine Neuerung ist die direkte Übersetzung der im UML-Modell definierten Daten- und Nachrichtenklassen anstatt, wie in Prosecco, die Verwendung eines rekursiven anwendungsunabhängigen Daten- und Nachrichtenformats. Das formale Modell ist außerdem nah am UML-Anwendungsmodell. Dadurch, dass die Abbildung der in UML modellierten Protokollschritte auf ASM-Regeln sehr einfach ist, muss die ASM bei der Verifikation nicht im Detail betrachtet werden. Stattdessen reicht es aus, sich das (besser lesbare) UML-Anwendungsmodell anzusehen. Außerdem ist die formale Spezifikation zugeschnitten auf die konkrete Anwendung. Wird zum Beispiel in einem UML-Modell keine Verschlüsselung verwendet, muss diese auch bei der Berechnung des Angreiferwissens nicht berücksichtigt werden. Auch das ist eine nicht zu unterschätzende Erleichterung bei der Durchführung der Verifikation.

Der SecureMDD-Ansatz berücksichtigt außerdem die Sicherheit auf Codeebene. Es ist sichergestellt, dass der automatisch generierte Anwendungscode korrekt, d.h. eine Verfeinerung des generierten formalen Modells ist. Dies bedeutet insbesondere, dass sich die auf dem formalen Modell bewiesenen Sicherheitseigenschaften auch auf den generierten Quellcode übertragen.

Der in dieser Arbeit vorgestellte Entwicklungsansatz ist vollständig toolunterstützt. Er wurde anhand von sechs verschiedenen Fallstudien evaluiert und weiterentwickelt. Auch die inkrementelle Entwicklung großer Fallstudien ist ohne Probleme möglich. Dies wurde anhand der Fallstudie der „Elektronischen Gesundheitskarte“ gezeigt. Dies ist der erste Ansatz, der für eine modellierte Smart Card-Anwendung lauffähigen Quellcode sowie ein formales Modell generiert, so dass sich die auf formaler Ebene verifizierten anwendungsspezifischen Sicherheitseigenschaften automatisch auf den generierte Anwendungscode übertragen.

Nach bestem Wissen ist dies die erste Arbeit, die einen modellgetriebenen Entwicklungsansatz zur Verfügung stellt, der den formalen Nachweis von (anwendungsspezifischen) Sicherheitseigenschaften für Smart Card-Anwendungen unterstützt und gleichzeitig lauffähigen Code generiert, für den die bewiesenen Eigenschaften ebenfalls gelten.

13.2. Ausblick

Die Ergebnisse dieser Dissertation dienen bereits als Grundlage für weitere Forschungsarbeiten. In einem Habilitationsvorhaben wird zurzeit erforscht, ob der in dieser Arbeit nur informell betrachtete Korrektheitsaspekt (siehe Kapitel 10) auch formal nachweisbar ist. Die Idee ist es, werkzeugunterstützt mit dem KIV-System zu verifizieren, dass der generierte Quellcode immer eine Verfeinerung des ebenfalls generierten formalen Modells ist. Für diesen Beweis müssen Aussagen über die implementierten Modelltransformationen gemacht werden. Hierfür wurde bereits ein Kalkül für die in SecureMDD verwendete Modell-zu-Modell-Transformationssprache QVT in das Beweissystem KIV integriert [181] und erste einfache Aussagen bewiesen. Das Vorhaben ist eine direkte Erweiterung dieser Arbeit.

Ein Vorteil des modellgetriebenen SecureMDD-Ansatzes ist die Erweiterbarkeit auf andere Anwendungsgebiete. Es gibt bereits ein Dissertationsvorhaben, das sich mit der Erweiterung des Ansatzes auf sichere Service-Anwendungen beschäftigt [23]. Dies ist eine sinnvolle Ergänzung dieser Arbeit, da einige Smart Card-Anwendungen ebenfalls Services nutzen. Das heißt, die Terminals einer Anwendung haben die Möglichkeit, mit Services zu kommunizieren und deren Operationen aufzurufen. Beispiele hierfür sind ein Smart Card-basiertes Ticketsystem für den Personennahverkehr, bei dem der Kauf über Services abgewickelt wird oder ein Online Banking-System, das für das Durchführen der Überweisungen Services verwendet. Eine weitere Erweiterung wäre die Unterstützung von Anwendungen für mobile Geräte, z.B. Smartphones.

Zurzeit werden die auf formaler Ebene zu verifizierenden Sicherheitseigenschaften im KIV-System als Theoreme formuliert. Dies ist momentan ein manueller Schritt. Alternativ wäre es ebenfalls möglich, die Sicherheitseigenschaften bereits im plattformunabhängigen UML-Modell zu formulieren. Geeignet hierfür wäre zum Beispiel die Object Constraint Language (OCL). Neben der automatischen Übersetzung hätte dies den Vorteil, dass die Sicherheitseigenschaften bereits formal im UML-Modell dokumentiert wären.

Eine weitere sinnvolle Ergänzung bezieht sich auf das in SecureMDD integrierte modellbasierte Testen der modellierten Anwendung. Das derzeitige Testframework bietet zwar bereits gute Unterstützung beim Testen des generierten Codes, kann aber noch erweitert werden. Wie bereits in Kapitel 7 beschrieben, sind hier die automatische Generierung von funktionalen Testfällen sowie die Einbindung einer Bibliothek mit Standardangriffen sinnvoll. Außerdem ist

das automatische Finden von Implementierungsfehlern, die unabhängig von der modellierten Anwendung sind, hilfreich. Dies kann zum Beispiel durch statische Checks des Quellcodes geschehen.

Zusammenfassend kann man sagen, dass diese Dissertation sowohl einen Beitrag zur aktuellen Forschung leistet als auch ein guter Grundstein für weitere Forschungsarbeiten ist. Insbesondere ist der modellgetriebene SecureMDD-Ansatz gut für die Erweiterbarkeit auf andere Anwendungsbereiche geeignet.

Teil VI.

Anhang

A Die plattformabhängigen Modelle einer Anwendung

Zusammenfassung: Das plattformunabhängige Modell einer Anwendung abstrahiert von den technischen Aspekten und Implementierungsdetails. Dies ermöglicht den Entwicklern, sich bei der Modellierung auf den Entwurf der eigentlichen Anwendung und der Protokolle zu konzentrieren. Dabei müssen sie sich keine Gedanken um die spätere Implementierung der modellierten Anwendung machen. Aus dem plattformunabhängigen Modell wird lauffähiger Code generiert. Um den Schritt vom abstrakten Modell hin zum Code für den Entwickler übersichtlicher und besser nachvollziehbar zu gestalten, wird zunächst ein plattformspezifisches Modell der Anwendung erzeugt. Für jede „Plattform“, also sowohl für die Smart Cards als auch die Terminals, wird ein eigenes plattformspezifisches Modell (PSM) generiert. Diese Zwischenebene ist eine grafische Repräsentation des Aufbaus und der Struktur des (später) generierten Codes und macht außerdem den Zusammenhang zwischen plattformunabhängigen Modell und Code deutlich. Um die Übersichtlichkeit zu erhöhen, sind jedoch einige weniger wichtige technische Details weggelassen. In diesem Kapitel wird der Aufbau der plattformspezifischen Modelle erläutert. Dies geschieht zunächst allgemein und anschließend anhand der Fallstudie der Kopierkartenanwendung.

In Kapitel 6 ist die Generierung des Quellcodes aus dem plattformunabhängigen UML-Modell beschrieben. Dabei sind die Modell-zu-Modell-Transformationen, die aus dem PIM die PSMs erzeugen, und die Modell-zu-Text-Transformationen, die aus den PSMs den Quelltext generieren, zu einem Transformationsschritt zusammengefasst. Der Vollständigkeit halber erläutert dieses Kapitel den Aufbau der plattformspezifischen Modelle.

Abschnitt A.1 beschreibt den Aufbau der plattformspezifischen Modelle im Allgemeinen, unabhängig von einer konkreten Anwendung. Die Abschnitte A.2 und A.3 illustrieren den Aufbau der PSMs am Beispiel der Kopierkartenanwendung.

In diesem Kapitel sind nur die Teile des plattformspezifischen Modells ausführlich erläutert, die sich gegenüber dem plattformunabhängigen Modell geändert haben oder neu hinzugekommen sind.

A.1. Aufbau allgemein

A.1.1. Klassendiagramme

Modellierung der Komponenten

Die Klassen, die die an der Anwendung beteiligten Komponenten modellieren, sind weiterhin im PSM enthalten. Handelt es sich um ein Terminal-PSM, werden alle Klassen mit Stereotyp ≪Smartcard≫ und ≪User≫ sowie die ausschließlich von diesen Komponenten verwendeten Daten- und Statusklassen entfernt. Analog werden im Smart Card-PSM die Klassen mit Stereotyp ≪Terminal≫ und ≪User≫ sowie die entsprechenden Daten- und Statusklassen entfernt. Auf Smart Card-Seite wird für eine Kapselung der Kommunikation mit dem Terminal eine zusätzliche Klasse mit dem Namen `SimpleComm` eingeführt. Diese enthält die Methoden für die Kommunikation mit dem Terminal und ist die Oberklasse der Klassen mit Stereotyp ≪Smartcard≫. Auf Terminalseite sind diese Methoden direkt in der Klasse mit Stereotyp ≪Terminal≫ implementiert. Die Klasse `SimpleComm` ist von der Klasse `javacard.framework.applet` abgeleitet. In Java Card müssen die Applets von dieser Klasse abgeleitet sein. Abbildung A.1 zeigt beispielhaft die Modellierung einer Smart Card-Klasse im Smart Card-PSM.

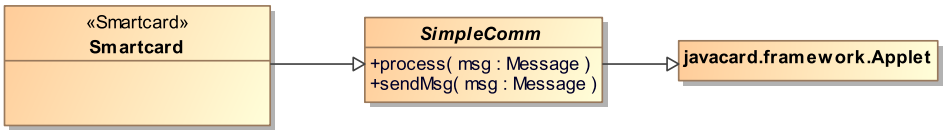

Abbildung A.1.: Eine Smart Card-Klasse im PSM

Die Methode `process(msg : Message)` wird beim Empfangen einer Nachricht (`msg`) aufgerufen und veranlasst die Verarbeitung dieser Nachricht, d.h. sie initiiert den entsprechenden Protokollschritt zum Verarbeiten der Nachricht. Die Methode `sendMsg(msg : Message)` sendet die Nachricht `msg` an die Terminalkomponente, von der zuvor eine Nachricht empfangen wurde. Da eine Smart Card nur auf eine empfangene Nachricht antworten kann, ist der Empfänger der Antwortnachricht bekannt und muss nicht explizit angegeben werden. Auf Terminalseite gibt es zusätzlich zu diesen beiden Methoden noch die Methode `sendMsg(msg : Message, port : int)`, die bei nicht eindeutigem Empfänger der Nachricht zusätzlich den Port angibt, über den die Nachricht gesendet wird (siehe Abschnitt 4.2.2).

Modellierung der Daten der Komponenten

Die Modellierung der Daten entspricht im Wesentlichen der Modellierung im plattformunabhängigen Modell. Allerdings werden die abstrakten primitiven Datentypen `Boolean`, `Number` und `String` durch primitive Datentypen der jeweiligen Zielsprache, also Java für das Terminal-PSM und Java Card für das Smart Card-PSM, ersetzt. Im Terminal-PSM gibt es die primitiven Datentypen `boolean`, `int`, `byte` und `String`. Der Typ `Number` aus dem plattformunabhängigen Modell wird auf den Typ `int` abgebildet. Da Java Card keine Strings und Integer unterstützt, unterscheiden sich die verwendeten Datentypen an dieser Stelle. Das

plattformspezifische Smart Card-Modell enthält die primitiven Typen `boolean`, `byte` und `short`. Der Typ `Number` wird auf Shortwerte abgebildet, Strings werden als Byte Arrays repräsentiert.

Da die Kryptographie in Java sowie Java Card auf Byte-Arrays operiert, sind die Sicherheitsdatentypen auf beiden Plattformen identisch. Bei allen Klassen der Sicherheitsdatentypen wurden die Attribute, die die eigentlichen Daten enthalten, und im plattformunabhängigen Modell vom Typ `String` sind, durch Byte-Arrays ersetzt. Weiterhin enthält jeder Sicherheitsdatentyp eine `copy`-Methode zum Kopieren eines Objekts sowie eine `equals`-Methode zum Test der Objektgleichheit zweier Objekte. Die in MEL vordefinierten kryptographischen Methoden, z.B. zum Generieren einer Nonce oder zum Verschlüsseln von Daten, sind in den plattformabhängigen Modellen bereits in die zuständigen Klassen der Sicherheitsdatentypen integriert. Die Methoden haben dasselbe Fehlerverhalten wie die entsprechenden MEL-Methoden (siehe Abschnitt 5.4).

Abbildung A.2 zeigt die Klassen `Secret` und `Nonce` sowie die Klassen für kryptographische Schlüssel.

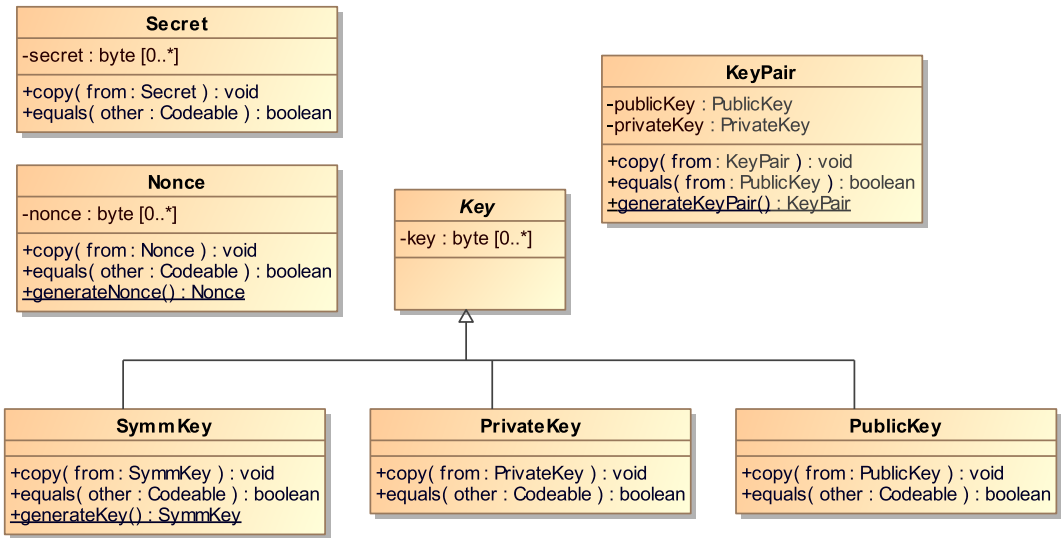

Abbildung A.2.: Ausschnitt aus den plattformspezifischen Sicherheitsdatentypen: Klassen Secret und Nonce sowie kryptographische Schlüssel

Die Klasse `Nonce` enthält die statische Methode `generateNonce()`, die eine neue Nonce zufällig erzeugt und zurückliefert.

Die Klasse `SymmKey` für symmetrische Schlüssel besitzt eine statische Methode `generateKey()`, die einen neuen, symmetrischen Schlüssel erzeugt und zurückliefert. Schlüsselpaare für asymmetrische Verschlüsselung werden durch die Klasse `KeyPair` repräsentiert. Sie besitzt eine statische Methode `generateKeyPair()`, die ein neues, zusammengehörendes Schlüsselpaar erzeugt und zurückliefert.

Abbildung A.3 zeigt die Klassen, die für die Speicherung verschlüsselter Daten sowie das Ver- und Entschlüsseln zuständig sind.

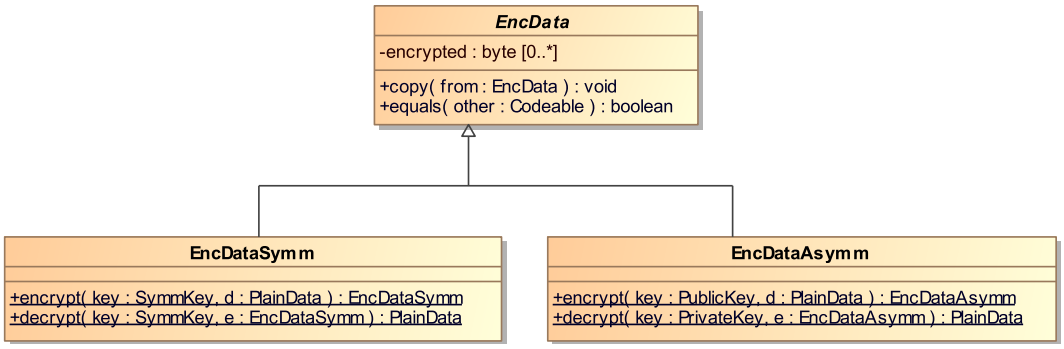

Abbildung A.3.: Ausschnitt aus den plattformspezifischen Sicherheitsdatentypen: Klassen zur Repräsentation verschlüsselter Daten

Die abstrakte Oberklasse `EncData` speichert das verschlüsselte Datum im Attribut `encrypted`. Wie im plattformunabhängigen Modell sind von dieser Klasse die Klassen `EncDataSymm` zur Speicherung symmetrisch verschlüsselter Daten sowie die Klasse `EncDataAsymm` zur Speicherung asymmetrisch verschlüsselter Daten abgeleitet. Die Klasse `EncDataSymm` besitzt die statische Methode `encrypt` zum symmetrischen Verschlüsseln von Daten. Diese erhält einen symmetrischen Schlüssel sowie ein unverschlüsseltes Objekt vom Typ `PlainData` als Eingabe und liefert ein Objekt vom Typ `EncDataSymm`, das die verschlüsselten Daten enthält, zurück. Die statische Methode `decrypt` entschlüsselt symmetrisch verschlüsselte Objekte. Als Eingabe enthält diese Methode einen symmetrischen Schlüssel sowie das verschlüsselte Objekt und liefert ein Objekt im Klartext, vom Typ `Plaindata`, zurück. Die Klasse `EncDataAsymm` enthält ebenfalls Methoden zum Ver- und Entschlüsseln. Diese sind analog definiert, operieren jedoch auf asymmetrischen Schlüsseln und Objekten vom Typ `EncDataAsymm`. Das Fehlerverhalten der Methoden ist gleich dem Verhalten der entsprechenden MEL-Methoden im Fehlerfall.

In Abbildung A.4 sind die Klassen zur Repräsentation von signierten und gehashten Daten sowie von MAC-Werten dargestellt.

Die Klasse `SignedData` speichert das signierte Objekt im Attribut `signed` vom Typ Byte Array. Die Klasse enthält die statische Methode `sign` zum Signieren von Datenobjekten. Die Eingabeparameter sind der private Schlüssel, mit dem das Objekt signiert wird, sowie das zu signierende Objekt vom Typ `PlainData`. Als Rückgabewert liefert die Methode ein neues Objekt vom Typ `SignedData`, das die signierten Daten enthält. Die statische Methode `verify` überprüft die Signatur eines Objekts vom Typ `SignedData`. Hierfür benötigt sie als Eingabe einen öffentlichen Schlüssel (mit dessen Gegenstück die Signatur erstellt wurde), das signierte Dokument vom Typ `SignedData` sowie das Dokument im Klartext vom Typ `SignData`. Als Rückgabe liefert die Methode einen booleschen Wert, der angibt, ob die signierten Daten den Klartextdaten entsprechen und ob das Dokument mit dem zu dem öffentlichen Schlüssel passenden privaten Schlüssel signiert wurde.

Die Klasse `HashedData` besitzt das Attribut `hashed` zum Speichern des Hashwertes, dieser ist vom Typ Byte-Array. Die Klasse besitzt die statische Methode `hash`, dessen Argument ein Objekt, über dem der Hashwert gebildet werden soll, ist. Als Rückgabe liefert diese Methode ein Objekt vom Typ `HashedData`, das die gehashten Daten enthält.

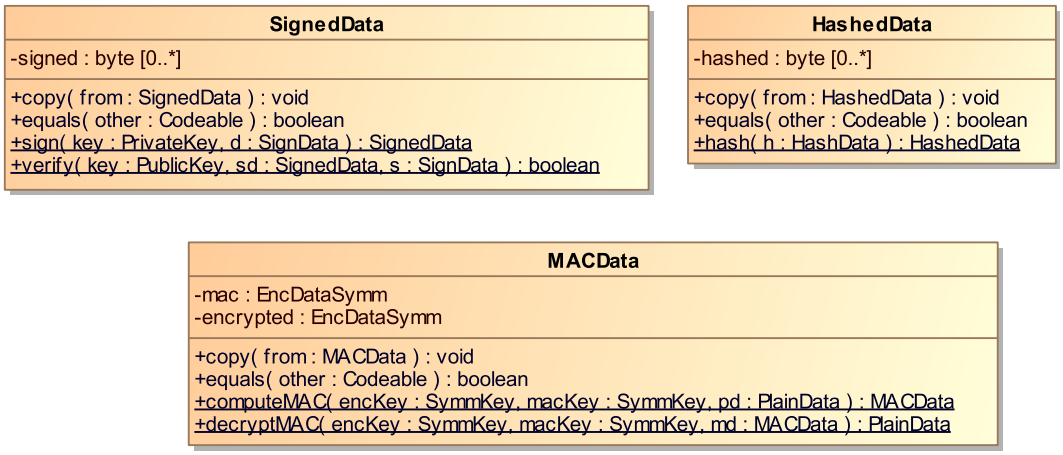

Abbildung A.4.: Ausschnitt aus den plattformspezifischen Sicherheitsdatentypen: Klassen zur Repräsentation signierter und gehashter Daten sowie von MAC-Werten

Die Klasse `MACData` repräsentiert den Message Authentication Code (MAC) einer Nachricht bzw. eines Objekts. Die Klasse `MACData` besitzt zwei Attribute. Das Attribut `mac` speichert den MAC-Wert, das Attribut `encrypted` enthält das Objekt in verschlüsselter Form. Die statische Methode `computeMAC` erhält als Argumente die beiden symmetrischen Schlüssel `encKey` und `macKey` sowie das Objekt, über dem der MAC-Wert gebildet werden soll (vom Typ `PlainData`). Die Methode verschlüsselt dann dieses Objekt mit dem Schlüssel `encKey` und speichert das Ergebnis im Attribut `encrypted`. Weiterhin wird der MAC-Wert unter Verwendung des Schlüssels `macKey` gebildet und im Attribut `mac` gespeichert. Als Rückgabewert liefert die Methode das Objekt vom Typ `MACData` zurück. Die Methode `decryptMAC` erhält als Eingabe beide symmetrische Schlüssel sowie ein Objekt vom Typ `MACData`. Die Methode entschlüsselt das im Attribut `encrypted` gespeicherte Dokument und gibt es zurück, falls der im Attribut `mac` gespeicherte MAC-Wert korrekt ist. Ansonsten bricht die Methode mit einer Fehlermeldung ab.

Kryptographische Daten, die Zustandsklasse und Nachrichtenklassen

Die Stereotypen ≪SignData≫, ≪HashData≫ und ≪PlainData≫ werden im plattformunabhängigen Modell verwendet, um die Klassen zu annotieren, deren Objekte während der Laufzeit der Anwendung signiert, gehasht bzw. verschlüsselt oder zur Berechnung des MAC-Wertes verwendet werden. Im plattformabhängigen Modell werden diese Stereotypen in Interfaces mit gleichem Namen übersetzt, die von den bisher annotierten Klassen implementiert werden. Die Stereotypen ≪encryptedSymm≫, ≪encryptedAsymm≫, ≪MAC≫, ≪hashed≫ und ≪signed≫ sind im plattformabhängigen Modell weiterhin vorhanden.

Die Modellierung des Zustands der Komponente ist gleich der Modellierung im plattformunabhängigen Modell.

Die Nachrichtenklassen sind gleich den Nachrichtenklassen im plattformunabhängigen Modell. Der einzige Unterschied ist, dass die abstrakte Oberklasse mit Stereotyp ≪Usermessage≫ als Subklasse der abstrakten Nachrichtenklasse mit Stereotyp ≪Message≫ modelliert ist. Der

Grund hierfür ist die Gleichbehandlung beider Nachrichtentypen im formalen Modell, die die formale Spezifikation und somit auch die Verifikation vereinfacht. Aus diesem Grund werden beide Nachrichtentypen auch in der Implementierung gleich behandelt. Wie im formalen Modell werden auch beim Verarbeiten und Versenden einer Nachricht dieselben Methoden für beide Nachrichtentypen verwendet.

Konstruktoren, manuelle Methoden sowie Konstantenklassen

Für jede Komponenten-, Daten- und Nachrichtenklasse im Klassendiagramm wird außerdem, falls noch nicht vorhanden, die Signatur für einen Konstruktor generiert. Für die Komponentenklassen ergeben sich die Parameter des Konstruktors aus den Attributen und Assoziationsenden der Klasse (sowie denen einer möglichen Oberklasse), die im plattformunabhängigen Modell den Stereotyp ≪Initialize≫ tragen. Für Daten- und Nachrichtenklassen enthält der Konstruktor Parameter für alle Attribute und Assoziationsenden der Klasse. Hat eine Klasse mehrere Assoziationen, muss bereits im plattformunabhängigen Modell ein Konstruktor definiert werden. Die Modellierung manueller Methoden entspricht der Modellierung im plattformunabhängigen Modell. Für die im PIM modellierten Konstanten, die in einer Konstantenklasse mit Stereotyp ≪Constant≫ zusammengefasst sind, werden im plattformspezifischen Modell Typangaben und Defaultwerte generiert. Konstanten können im PIM den Typ `Boolean`, `Number` oder `String` haben. Ist im PIM für eine Konstante ein Typ angegeben, wird der in den entsprechenden Typ der Zielplattform übersetzt. Ist kein Typ angegeben, wird der Typ `Number` verwendet. Boolesche Konstanten, für die im PIM kein Wert angegeben wurde, bekommen den Wert *false*. Konstanten vom Typ `Number` werden aufsteigend, von eins an, nummeriert. Für Konstanten vom Typ `String` muss bereits im PIM ein Wert angegeben werden.

Serialisierung und Deserialisierung von Datenklassen

Die Kommunikation mit einer Smart Card erfolgt vom Terminal aus über sogenannte APDUs (Application Protocol Data Unit). Eine APDU ist ein Byte-Array, das in einem vordefinierten Format vorliegen muss. Damit ein Terminal mit einer Smart Card kommunizieren kann, müssen die Daten, die gesendet werden sollen, serialisiert werden. In SecureMDD wird deshalb eine Kodierungskomponente, die verschickte Objekte in Byte-Arrays serialisiert bzw. nach dem Empfangen wieder deserialisiert, automatisch generiert. Da sowohl das Terminal als auch die Smart Card Daten serialisieren sowie deserialisieren müssen, ist diese Kodierungskomponente für beide plattformspezifische Modelle gleich. Abbildung A.5 zeigt die Klassen des PSMs, die für die Kodierung der Datenklassen und die Serialisierung zuständig sind. Diese Klassen werden im PSM hinzugefügt, im plattformunabhängigen Modell ist dieser technische Aspekt nicht modelliert.

Die in SecureMDD generierte Serialisierung basiert auf einer Typ-Length-Value (TLV) Kodierung, die von den ASN1 Kodierungsregeln für beliebige Daten [52] inspiriert ist. Das erste Byte enthält ein Typflag, das den Typ des serialisierten Objekts bestimmt. Die folgenden zwei Bytes enthalten die Länge des Objekts gefolgt von den eigentlichen Daten (repräsentiert als Byte-Array). Die Serialisierung erfolgt rekursiv über die Attribute einer Klasse, gefolgt von den assoziierten Klassen. Für alle Klassen, die zur Laufzeit serialisiert werden, benötigt man also ein Typflag. Die Klasse `Code` enthält diese Flags. Neben den zu serialisierenden

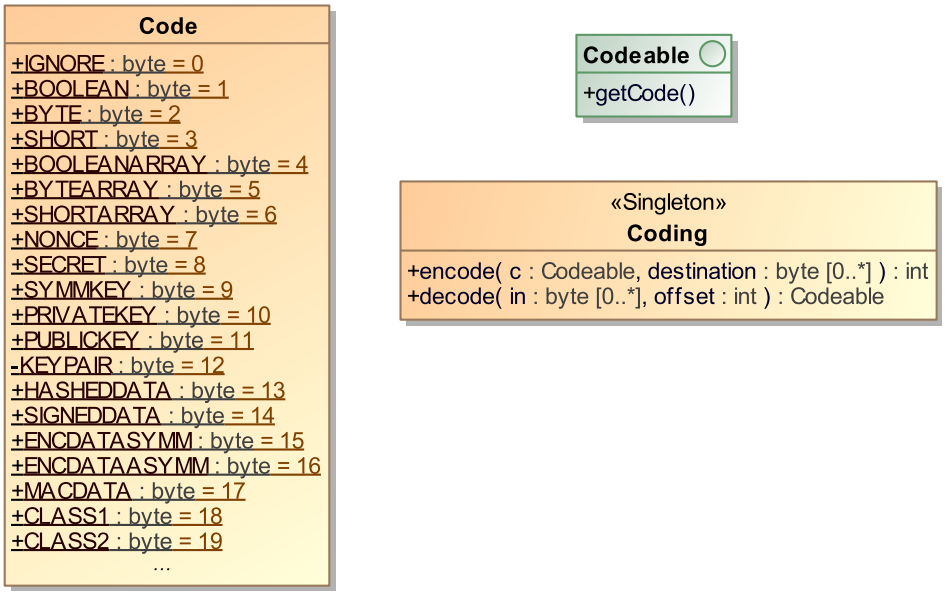

Abbildung A.5.: Klassen für die Serialisierung und Deserialisierung der zwischen Smart Card und Terminal ausgetauschten Daten

Datenklassen (`CLASS1`, `CLASS2`,...) werden Typflags für die primitiven Datentypen (zur Serialisierung der Attribute) sowie die Sicherheitsdatentypen generiert. Alle Flags sind vom Typ `byte` und von 0 an durchnummeriert. Zusätzlich wird das Flag `IGNORE` generiert, das für die Initialisierung einer Smart Card von Bedeutung ist. Weiterhin implementieren alle serialisierbaren Klassen das Interface `Codeable`. Dieses gibt an, dass die entsprechende Klasse kodierbar, d.h. serialisierbar, ist. Das Interface definiert eine Methode `getCode()`, die den Typflag der Klasse zurückgibt. Die Klasse `Coding` enthält die eigentlichen Methoden zum Kodieren und Dekodieren der Daten. Sie ist als Singleton realisiert. Die Methode `encode` erhält das zu serialisierende Objekt (vom Typ `Codeable`) sowie das Byte-Array, in das das Ergebnis geschrieben werden soll und liefert die Länge des serialisierten Objekts zurück. Die Methode `decode` erhält das Byte Array `in`, in dem das serialisierte Objekt gespeichert ist sowie die Stelle, ab dem das Objekt im Array gespeichert ist (`offset`). Als Ergebnis liefert die Methode ein Objekt vom Typ `Codeable` zurück. Innerhalb der Methode werden rekursiv die zu dem Objekt gehörenden Attribute und assoziierten Klassen deserialisiert.

Speicherverwaltung für Smart Cards

Eine Besonderheit bei der Programmierung von Java Card Smart Cards ist die fehlende Garbage Collection auf der Karte. Dies bedeutet, dass der Programmierer achtsam bei der Allokation von Speicher sein muss, da dieser nicht mehr freigegeben wird und gleichzeitig das Speichervolumen auf der Karte begrenzt ist. Der SecureMDD-Ansatz sieht zur Lösung dieses Problems eine automatische Objektverwaltung vor. Diese analysiert auf Basis des plattformunabhängigen Modells zunächst, wie viele Objekte der modellierten Klassen tatsächlich zur Laufzeit benötigt werden. Diese werden dann bei der Installation des Applets erzeugt und

vom Objektmanager verwaltet. Wird auf der Smart Card in einem Protokollschritt ein neues Objekt benötigt, wird dieses nicht neu erzeugt, sondern von dem Objektmanager herausgegeben. Am Ende des Protokollschritts, wenn das Objekt nicht mehr benötigt wird, wird es an diese wieder zurückgegeben. Der Objektmanager wird automatisch generiert. Das plattformspezifische Java Card-Modell enthält die Klasse `Store`, die für die Objektverwaltung zuständig ist. Diese ist in Abbildung A.6 dargestellt.

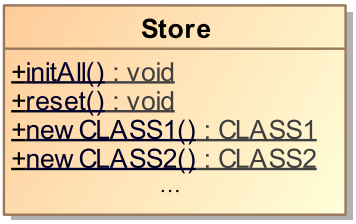

Abbildung A.6.: Klasse Store für die Objektverwaltung

Die statische Methode `initAll()` beinhaltet die Initialisierung. Sie erzeugt alle während der Protokollläufe benötigten Objekte und speichert sie in der Klasse `Store`. Die Methode wird bei Installation des Applets aufgerufen. Die statische Methode `reset()` setzt den Store auf den Anfangszustand zurück. Dies ist gleichbedeutend mit der Rückgabe aller ausgegebenen Objekte an den `Store` und darf nur erfolgen, wenn die Smart Card-Klasse (d.h. die Appletklasse) die von der Speicherverwaltung herausgegebenen Objekte nicht mehr benötigt. Die Methode wird am Ende eines Protokollschritts aufgerufen. Die Methoden `newCLASS1()` und `newCLASS2()` werden aufgerufen, wenn die Smart Card-Klasse ein neues Objekt vom Typ `CLASS1` bzw. `CLASS2` benötigt. Die Methode stellt diese Objekte zur Verfügung, sie können von der Smart Card bis zum Ende des Protokollschritts verwendet werden. Für jedes von dem Objektmanager verwaltete Objekt wird eine `new`-Methode erzeugt, die ein Objekt dieser Klasse zur Verfügung stellt.

A.1.2. Aktivitätsdiagramme

Die Aktivitätsdiagramme der plattformspezifischen Modelle unterscheiden sich von denen der plattformunabhängigen Modelle. Im plattformspezifischen Modell enthalten die Aktivitätsdiagramme nur noch Aktivitätsbereiche für die Komponenten der Plattform. Die Aktivitätsdiagramme des Terminal-PSMs enthalten nur die Aktivitätsbereiche für die Terminals, die Bereiche für den Benutzer und die Smart Cards werden entfernt. Im Smart Card-PSM sind nur die Aktivitätsbereiche für die Smart Cards gespeichert.

Ist im plattformunabhängigen Modell ein Aktivitätsbereich für eine abstrakte Oberklasse definiert, wird das Aktivitätsdiagramm im PSM entsprechend der Anzahl der von dieser Klasse abgeleiteten Komponentenklassen vervielfacht. Ein Beispiel hierfür ist das Abfragen des Kontostands im Terminal-PSM. Das Abfragen des Kontostands ist sowohl am Ladeterminal als auch am Kopiergerät möglich. Um im PIM das Protokoll nur einmal modellieren zu müssen, wurde ein Aktivitätsbereich für die abstrakte Oberklasse `Terminal` modelliert. Im PSM wird dieses Aktivitätsdiagramm dupliziert. Ein Diagramm enthält dann den Aktivitätsbereich für das Ladeterminal, das andere Diagramm den für das Kopiergerät (siehe A.2.2).

Jedes Aktivitätsdiagramm enthält im PSM genau einen Aktivitätsbereich. Sind an einem Protokoll mehrere Komponenten desselben Typs, d.h. mehrere Terminals oder mehrere Smart Cards beteiligt, gibt es im PSM für jede Komponente ein eigenes Aktivitätsdiagramm. In einem Aktivitätsbereich sind, wie im PIM auch, alle Schritte eines Protokolls einer Komponente dargestellt.

Die DSL MEL, die in der plattformunabhängigen Modellierung verwendet wird, wird in den plattformspezifischen Modellen durch die Programmiersprache der jeweiligen Plattform ersetzt. Die Aktivitätsdiagramme des Terminal-PSMs enthalten Java-Code, die Diagramme des Smart Card-PSMs enthalten Java Card-Code. Dieser Code wird bei der Generierung des Quellcodes aus den PSMs übernommen, d.h. der in den UML-Elementen der PSMs stehende Java- bzw. Java Card-Code findet sich genauso im lauffähigen Quellcode wieder.

Im Terminal-PSM werden die Attribute und Assoziationsenden der Daten-, Nachrichten-, und Komponentenklassen als `private`, d.h. von außerhalb der Klasse nicht sichtbar, angesehen. Auf sie kann mittels `get`- und `set`-Methoden zugegriffen werden. Im Smart Card-PSM werden die Attribute und Assoziationsenden als `public`, d.h. für alle anderen Klassen auch sichtbar, angesehen. Deshalb kann auf die Attribute direkt zugegriffen werden. Die Generierung von `get`- und `set`-Methoden dient der Kapselung und besseren Wartbarkeit. Für die Terminals ist die Verwendung sinnvoll, da auch externe Komponenten, zum Beispiel eine graphische Benutzeroberfläche, auf die Daten zugreifen möchten. Die Kartenkomponente dagegen befindet sich in einer abgeschlossenen Umgebung (d.h. auf einer Smart Card). Der Schutz der Attribute und Assoziationsenden durch `get`- und `set`-Methoden ist deshalb nicht notwendig.

Im Folgenden wird auf die Besonderheiten der in den UML-Elementen enthaltenen Java- bzw. Java Card-Ausdrücke eingegangen:

- UML-`SendSignalAction`: Das Senden einer Nachricht ist auch im PSM durch eine UML-`SendSignalAction` modelliert. Diese enthält den Aufruf der `sendMsg`-Methode, die im Code beim Versenden einer Nachricht aufgerufen wird. Als Argumente bekommt die Methode die zu sendende Nachricht sowie, wenn notwendig, die Angabe des Ports, über den die Nachricht gesendet werden soll.
- UML-`AcceptEventAction`: Das Empfangen einer Nachricht ist auch im PSM durch eine UML-`AcceptEventAction` modelliert. Diese enthält den Aufruf der `process`-Methode für den jeweiligen Nachrichtentyp. Für jeden Nachrichtentyp wird eine eigene `process`-Methode generiert, die die Implementierung des gesamten Protokollschritts enthält. Hat der Nachrichtentyp Attribute und Assoziationen, werden die darin gespeicherten Daten der Nachricht in lokalen Variablen gespeichert. In diesem Fall folgt auf die UML-`AcceptEventAction` eine UML-`Action`, die lokale Variablen für die Attribute und Assoziationsenden der Nachrichtenklasse deklariert und mit den in der Nachricht enthaltenen Daten initialisiert. Die Namen der lokalen Variablen ergeben sich aus dem entsprechenden MEL-`Receive` Ausdruck im PIM.
- UML-`DecisionNode`: Die UML-`DecisionNodes` werden aus dem PIM übernommen, die MEL-Ausdrücke der Guards werden jedoch in Java- bzw. Java Card-Ausdrücke übersetzt. Eine Besonderheit hier ist, dass das in MEL definierte Schlüsselwort `else` nicht übersetzt wird.

- UML-`FlowFinalNode`: Ein `FlowFinalNode` modelliert im PIM das Auftreten eines Fehlers, der zum Abbruch des Protokolllaufs führt. Im PSM bleiben sind die `FlowFinalNodes` weiterhin vorhanden. Sie beenden den aktuellen Protokollschritt und Werfen eine SecureMDD-Exception.
- Subdiagrammaufrufe: Subdiagramme werden im lauffähigen Quellcode durch Methoden implementiert, d.h. ein Subdiagramm entspricht einer Methode, die in der Komponentenklasse, für die das Subdiagramm definiert ist, implementiert ist. Aus den `ActivityParameterNodes` werden die Parameter dieser Methode generiert. Der Aufruf eines Subdiagramms entspricht dann im PSM dem Aufruf der entsprechenden Methode.

Die weiteren im PIM verwendeten UML-Elemente bleiben im PSM unverändert.

A.1.3. Deploymentdiagramme

Die plattformunabhängigen Deploymentdiagramme beschreiben die Kommunikationsstruktur der Anwendung sowie die Möglichkeiten des Angreifers (Lesen, Schreiben und Unterdrücken einer Nachricht). Die Deploymentdiagramme der PSMs sind gleich den Diagrammen im PIM. Die Informationen über die Möglichkeiten des Angreifers werden für die Generierung des Quellcodes nicht benötigt. Da die Festlegungen, die für die Angreiferfähigkeiten im Deploymentdiagramm gemacht werden, jedoch das Design der Protokolle beeinflussen, ist diese Information in den PSMs weiterhin enthalten.

Zurzeit wird das Importieren eines transformierten UML-Modells in den UML-Modellierungstools nur begrenzt unterstützt. Das Problem ist, dass die graphischen Informationen eines UML-Modells beim Exportieren verloren gehen. Aus diesem Grund lässt sich ein transformiertes Modell, d.h. jedes PSM im SecureMDD-Ansatz, im Modellierungstool zwar wieder importieren, jedoch nicht graphisch anzeigen. Für Klassendiagramme ist dies im Modellierungstool Magic Draw durch einen Klassendiagramm-Wizard gelöst, der die UML-Elemente graphisch anordnet und darstellt. Für Deployment- und Aktivitätsdiagramme gibt es diese Unterstützung jedoch momentan noch nicht. Das graphische Anzeigen dieser transformierten Diagramme ist somit zurzeit sehr umständlich und wenig praktikabel. Dies kann sich jedoch in einer späteren Version der Software ändern.

A.2. Beispiel Kopierkartenanwendung: Das Terminal-PSM

Dieser Abschnitt beschreibt das plattformspezifische Modell der Terminals am Beispiel der Kopierkartenanwendung.

A.2.1. Klassendiagramme

Abbildung A.7 zeigt die abstrakte Klasse `Terminal` sowie die konkreten Terminalklassen `CopyingMachine` und `DepositMachine`.

Die abstrakte Terminalklasse besitzt, wie auch im plattformunabhängigen Modell, das Attribut `passphrase` vom Typ `Secret` sowie eine Assoziation zur Klasse `StateTerminal`,

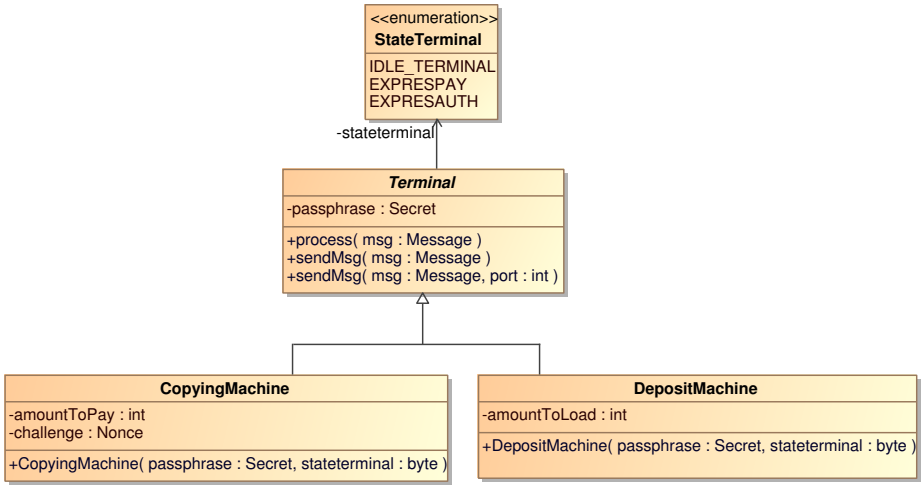

Abbildung A.7.: Ausschnitt aus dem plattformspezifischen Terminal-Modell: Klassen für die Terminals und assoziierte Klassen

die die möglichen Zustände eines Terminals speichert. Das Assoziationsende ist mit Stereotyp ≪status≫ annotiert. In der Klasse `Terminal` sind außerdem die Operationen zum Senden und Empfangen einer Nachricht definiert.

In den Klassen `CopyingMachine` und `DepositMachine` sind die Typen der Attribute durch Java-Datentypen ersetzt worden. Die Attribute `amountToPay` und `amountToLoad` haben den Typ `int` (vorher `Number`). Das Attribut `challenge` ist weiterhin vom Typ `Nonce`. Zusätzlich wird für jede Klasse ein Konstruktor, der Parameter für alle Attribute und Assoziationsenden mit Stereotyp ≪Initialize≫ besitzt, definiert. In diesem Beispiel haben beide Konstruktoren einen Parameter vom Typ `Secret` (für das Attribut `passphrase`) sowie einen vom Typ `byte` (für den Status).

Neben den Terminalklassen sind auch die Nachrichtenklassen im plattformspezifischen Modell vorhanden. Diese sind in Abbildung A.8 dargestellt.

Die Benutzernachrichten und die Nachrichten, die zwischen einem Terminal und einer Smart Card ausgetauscht werden, werden gleichbehandelt. Deshalb ist die abstrakte Klasse `Usermessage`, die die Superklasse aller Benutzernachrichtenklassen ist, von der Klasse `Message` abgeleitet. Alle Nachrichten müssen, um sie an eine Smart Card schicken zu können, serialisierbar sein und implementieren deshalb das Interface `Codeable`.

Die Nachrichtenklassen entsprechen denen im plattformunabhängigen Modell. Die Typen der Attribute aller Klassen sind durch Java-Datentypen ersetzt worden. Die Klasse `AuthData`, die im PIM mit Stereotyp ≪HashData≫ annotiert ist, implementiert im PSM das Interface `HashData`. Der Stereotyp ≪hashed≫ annotiert auch im PSM die Assoziationsenden `authterminal` und `authcard`. Zusätzlich besitzt jede Nachrichten- und Datenklasse im PSM einen Konstruktor, der für alle Attribute und Assoziationsenden einen Parameter vom passenden Typ besitzt.

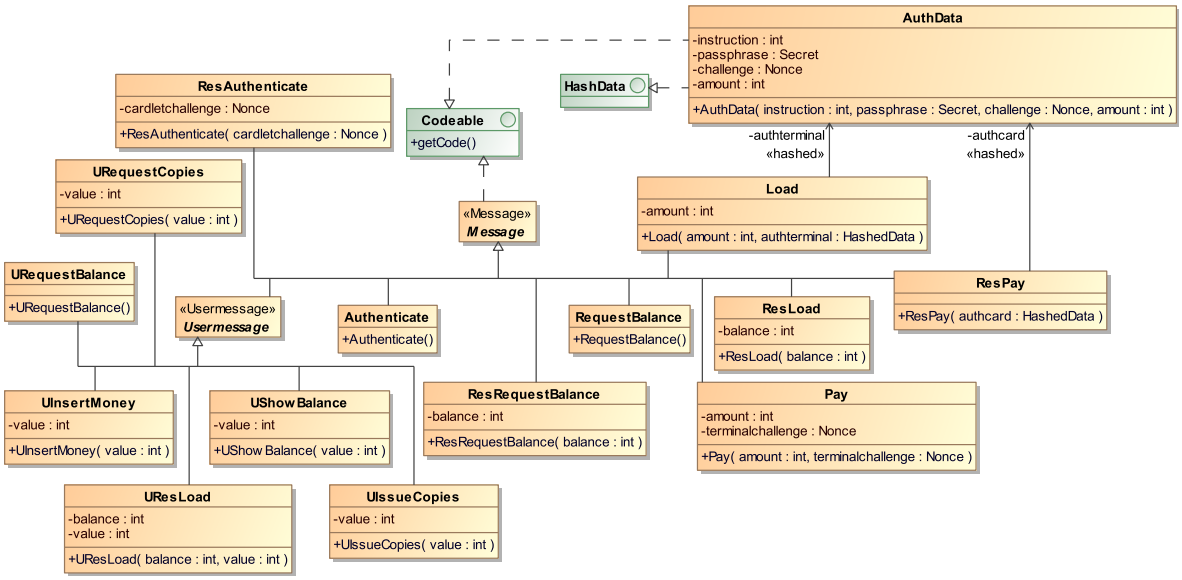

Abbildung A.8.: Ausschnitt aus dem plattformspezifischen Terminal-Modell: Klassen für die Nachrichten und assoziierte Klassen

Abbildung A.9 zeigt die verbleibenden, noch nicht erläuterten Klassen des Terminal-PSMs. Dies sind die Klassen `Coding`, `Code` und `Constants`.

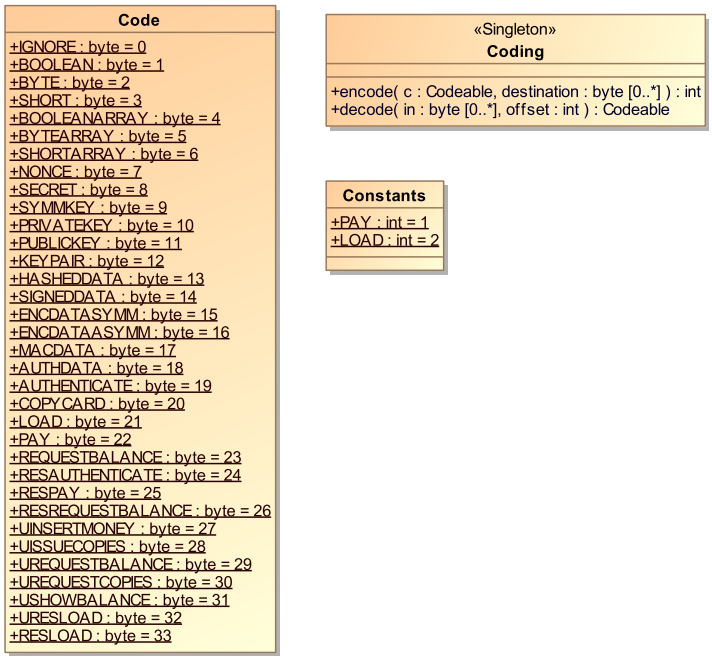

Abbildung A.9.: Ausschnitt aus dem plattformspezifischen Terminal-Modell: Klassen `Code`, `Coding` und `Constants`

Die Klasse `Coding` enthält die Methoden zum Serialisieren und Deserialisieren von Objekten. Da nur eine Instanz dieser Klasse benötigt wird, ist sie als `Singleton` implementiert.

Die Klasse `Code` wird ebenfalls für die (De-)Serialisierung von Objekten benötigt. Um die als Byte-Array repräsentierten Daten wieder in ein Objekt zu deserialisieren, ist es notwendig sich den Typ des Objekts zu merken. Für jede zu serialisierende Klasse ist hierfür eine Typflag vom Typ `Byte` definiert. Diese sind in der Klasse `Code` zusammengefasst.

Die Klasse `Constants` enthält die im plattformunabhängigen Modell definierten Konstanten. Dort war für die Konstanten `PAY` und `LOAD` kein Typ und kein Defaultwert angegeben. Diese werden bei Generierung des PSMs ergänzt. Beide Konstanten bekommen im Terminal-PSM den Typ `int` und werden von eins an aufsteigend nummeriert.

A.2.2. Aktivitätsdiagramme

Im Folgenden werden die Aktivitätsdiagramme des plattformspezifischen Terminal-Modells erläutert.

Abfragen des Kontostands
Abbildung A.10 zeigt das Aktivitätsdiagramm für das Abfragen des Kontostands der Karte an einem Kopiergerät. Das Protokoll ist für beide Terminalarten, also die `CopyingMachine` und die `DepositMachine`, gleich. Im plattformunabhängigen Modell ist deshalb nur ein Aktivitätsdiagramm für beide Terminalarten definiert, der entsprechende Aktivitätsbereich trägt den Namen der gemeinsamen Oberklasse, `Terminal`. Im plattformspezifischen Modell wird jeweils ein eigenes Aktivitätsdiagramm für jede konkrete Terminalinstanz generiert. Der Übersichtlichkeit halber ist hier nur das Aktivitätsdiagramm für das Kopiergerät abgebildet.

Das Aktivitätsdiagramm enthält den Aktivitätsbereich für das Kopiergerät (Klasse `CopyingMachine`). Hintereinander sind hier die einzelnen Protokollschritte, die das Kopiergerät ausführt, dargestellt. Die Aktivitätsbereiche der anderen beteiligten Komponenten werden bei der Generierung des PSMs entfernt. Die UML-Elemente innerhalb der Aktivitätsbereiche enthalten Java-Code, der aus den MEL-Ausdrücken generiert wurde, und bei der Codegenerierung unverändert übernommen wird.

Beim Abfragen des Kontostands führt das Kopiergerät zwei Protokollschritte durch. Der erste Protokollschritt beginnt mit dem Empfangen einer `URequestBalance`-Nachricht. Wie im PIM ist dies durch eine UML-`AcceptEventAction` modelliert. Für jeden Nachrichtentyp besitzt die Klasse `CopyingMachine` eine eigene `process`-Methode, die beim Empfang aufgerufen wird und für die Verarbeitung der Nachricht verantwortlich ist. Das Empfangen der Nachricht ist deshalb durch den Aufruf der `processURequestBalance`-Methode, die ein Objekt `inmsg` vom Typ `URequestBalance` als Argument bekommt, modelliert (1). Nach Empfang dieser Nachricht wird das Attribut `stateterminal`, das den Zustand des Kopiergeräts repräsentiert, auf `IDLE_TERMINAL` gesetzt. Hierfür wird in Java die entsprechende `set`-Methode aufgerufen (2). Das Senden der `RequestBalance()`-Nachricht an die Kopierkarte ist im PSM ebenfalls durch eine `SendSignalAction` modelliert. Diese enthält einen Aufruf der Methode `sendMsg`, das Argument der Methode ist ein neu erzeugtes Objekt vom Typ `RequestBalance` (3). Da es nur eine Smart Card-Komponente in der Anwendung gibt und der Empfänger der Nachricht somit eindeutig ist, wird kein Port angegeben.

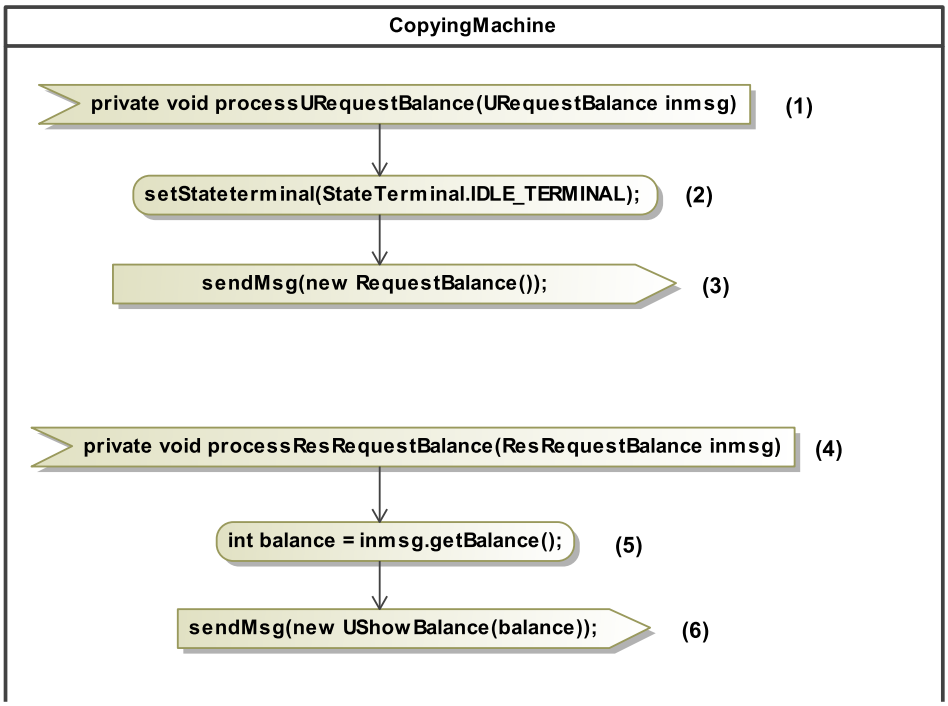

Abbildung A.10.: Ausschnitt aus dem plattformspezifischen Terminal-Modell: Abfragen des Kontostands

Der nächste Protokollschritt beginnt mit dem Empfangen der `ResRequestBalance`-Nachricht. Dies ist innerhalb der `AcceptEventAction` modelliert durch den Aufruf der `processResRequestBalance`-Methode (4). Die `ResRequestBalance`-Klasse hat ein Attribut mit dem Namen `balance` vom Typ `int`. Im plattformunabhängigen Modell wird für jedes Attribut und Assoziationsende der Nachrichtenklasse implizit eine lokale Variable deklariert, die innerhalb des aktuellen Protokollschrittes verwendet werden kann (siehe Abschnitt 4.4.2.2). Im PSM geschieht die Deklaration der lokalen Variablen explizit in einer `Action` (5). Die lokale Variable hat, wie im PIM, den Namen `balance` und bekommt den in der Nachricht enthaltenen Wert zugewiesen. Auf diesen wird über den Ausdruck `inmsg.getBalance` zugegriffen. Der Protokollschritt endet mit dem Senden einer `UShowBalance`-Nachricht, die mit dem Wert der lokalen Variable `balance` als Argument neu erzeugt wird (6).

Aufladen der Kopierkarte:
Abbildung A.11 zeigt das Aktivitätsdiagramm des Terminal-PSMs zum Laden von Geld auf eine Kopierkarte.

Da am Aufladen einer Kopierkarte ein Ladeterminal beteiligt ist, enthält das Aktivitätsdiagramm einen Aktivitätsbereich für die `DepositMachine`. Das Protokoll zum Aufladen startet auf Seite der `DepositMachine` mit dem Empfangen und Verarbeiten einer `UInsertMoney`-Nachricht. Dies geschieht durch Aufruf der `processUInsertMoney`-Methode (1). In der nachfolgenden `Action` wird eine lokale Variable `val` für den in der `UInsertMoney`-Nachricht enthaltenen Wert deklariert (2). Wie auch im PIM ist ein if-then-else durch UML-`DecisionNodes` modelliert. Ist der Wert `val` kleiner oder gleich Null, wird

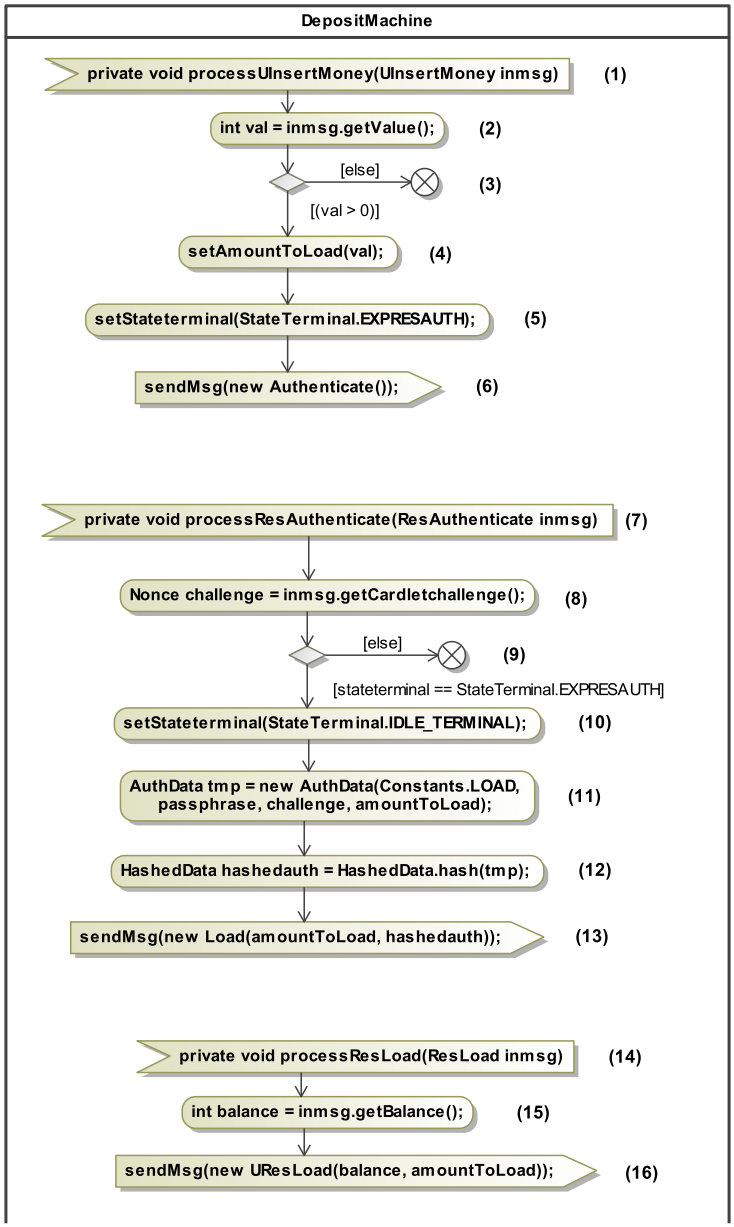

Abbildung A.11.: Ausschnitt aus dem plattformspezifischen Terminal-Modell: Aktivitätsdiagramm für das Aufladen der Karte

das Protokoll abgebrochen (3). Dies ist, wie auch im PIM, durch einen `FlowFinal`-Knoten modelliert und führt im generierten Code zum Werfen einer Exception. Anderenfalls wird das Protokoll fortgeführt. Der Wert der Variablen `val` wird durch Aufruf der entsprechen-

den set-Methode der Variablen amountToLoad zugewiesen (4). Als nächstes wird der Zustand des Ladeterminals auf EXPRESAUTH gesetzt (5) und anschließend eine neu erzeugte Authenticate-Nachricht versendet (6). Dies geschieht durch Aufruf der sendMsg-Methode.

Der nächste Protokollschritt beginnt mit dem Empfang einer ResAuthenticate-Nachricht bzw. dem Aufruf der processResAuthenticate-Methode auf Seiten des Ladeterminals (7). Nachdem für die in der Nachricht enthaltenen Nonce die lokale Variable challenge deklariert wurde (8), wird überprüft, ob der Zustand des Ladeterminals gleich EXPRESAUTH ist (9). Ist dies nicht der Fall, bricht das Protokoll mit einem FlowFinalNode ab. Ansonsten wird der Zustand stateterminal des Ladeterminals auf IDLE_TERMINAL gesetzt (10). Als nächstes wird eine lokale Variable vom Typ AuthData erzeugt. Auch hier wird Java-Syntax verwendet und das Objekt mit dem Schlüsselwort new erzeugt (11). Darauffolgend wird über dem soeben erzeugten Objekt tmp der Hashwert gebildet und der neu deklarierten lokalen Variablen hashedauth zugewiesen (12). Anders als im PIM wird nicht die MEL-Methode hash() aufgerufen, sondern die entsprechende Java-Methode, die in der Klasse HashedData implementiert ist. Der Protokollschritt endet mit dem Senden einer Load-Nachricht. Dies ist modelliert durch den Aufruf der Methode sendMsg. Als Argument erhält diese ein neues Load-Objekt, das mit dem Wert des Attributs amountToLoad sowie dem in der Variablen hashedauth gespeicherten Hashwert erzeugt wird (13).

Der letzte Protokollschritt beginnt mit dem Empfang einer ResLoad-Nachricht (14). Anschließend wird eine lokale Variable balance für den in der Nachricht enthaltenen Kontostand der Karte deklariert und mit dem in der Nachricht enthaltenen Wert initialisiert (15). Zum Abschluss des Protokolls wird eine UResLoad-Nachricht erzeugt und mittels der sendMsg-Methode versendet.

Anfertigen von Kopien
In Abbildung A.12 ist das Aktivitätsdiagramm des Terminal-PSMs für das Anfertigen von Kopien dargestellt.

Der erste Protokollschritt beginnt mit dem Empfang einer URequestCopies-Nachricht bzw. dem Aufruf der processURequestCopies-Methode in der Klasse CopyingMachine (1). Für das Attribut value des Objekts vom Typ URequestCopies (das den zu zahlenden Betrag enthält) wird anschließend die lokale Variable val deklariert und ihr der in der Nachricht enthaltene Wert zugewiesen (2). Als nächstes wird überprüft, ob dieser Wert größer als null ist (3). Im negativen Fall bricht das Protokoll ab. Ist die Bedingung erfüllt, wird durch Aufrufen der set-Methode des Attributs amountToPay dieser Wert auf val gesetzt (4). Im Anschluss daran wird eine neue Nonce erzeugt und im Attribut challenge gespeichert (5). Im PIM wurde zum Erzeugen einer neuen Nonce die MEL-Methode generateNonce() aufgerufen. Im PSM ist dieser Aufruf durch den Aufruf einer entsprechenden Java Card-Methode ersetzt worden. Diese ist in der Klasse Nonce deklariert. Aufgrund der Tatsache, dass die Kopiersemantik von MEL auch für den generierten Code gelten muss (siehe Abschnitt 10.3.4), muss die neu generierte Nonce an dieser Stelle kopiert werden. Um Objekte kopieren zu können, implementiert jede Klasse die copy()-Methode. Diese wird nun verwendet, um die neu erzeugte Nonce zu duplizieren und dem Attribut challenge zuzuweisen. Anschließend wird der Zustand des Terminals durch Aufruf der entsprechenden set-Methode auf EXPRESPAY gesetzt (6) sowie eine neue Pay-Nachricht erzeugt und durch Aufruf der sendMsg()-Methode versendet (7).

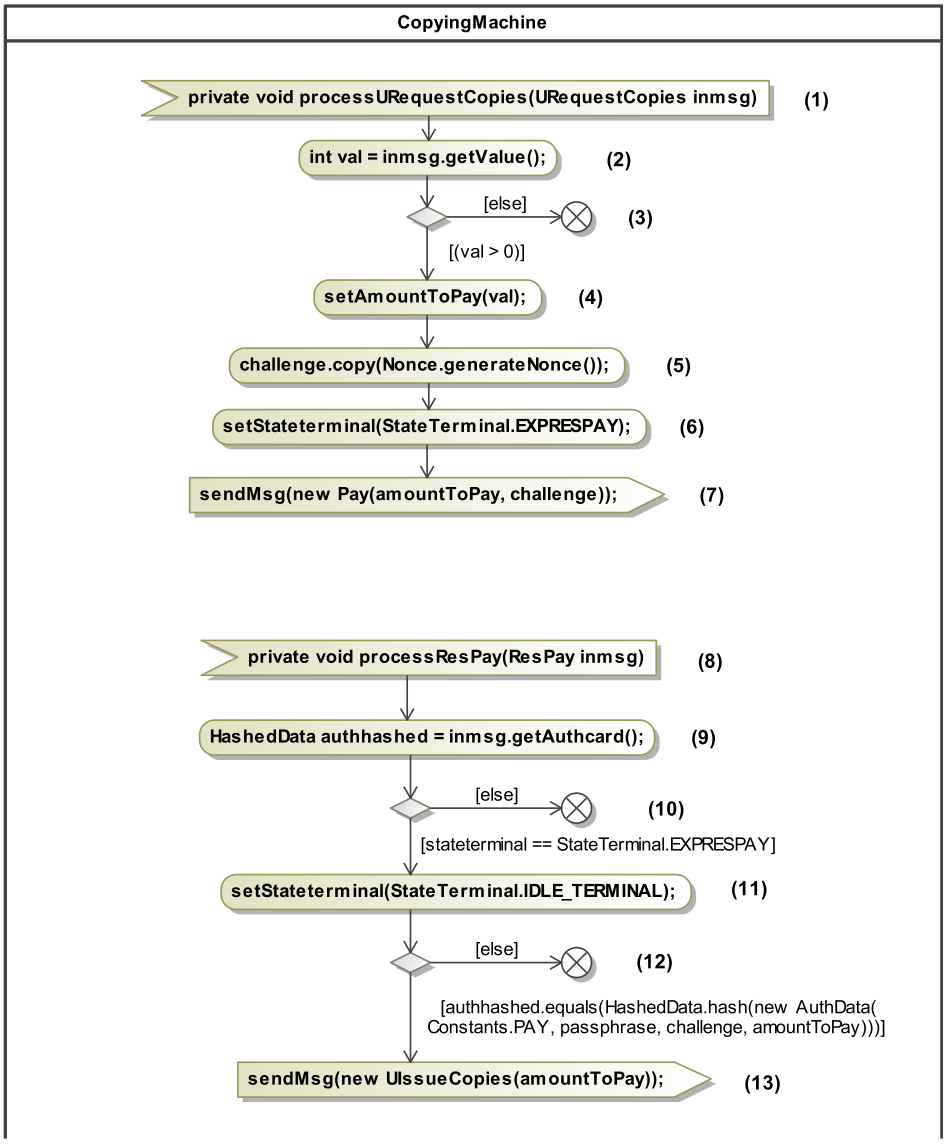

Abbildung A.12.: Ausschnitt aus dem plattformspezifischen Terminal-Modell: Aktivitätsdiagramm für das Anfertigen von Kopien

Der letzte Protokollschritt beginnt mit dem Empfang einer `ResPay`-Nachricht bzw. dem Aufruf der `processResPay`-Methode in der Klasse `CopyingMachine` (8). Anschließend wird der in der Nachricht enthaltene Hashwert in der neu deklarierten lokalen Variablen `authhashed` gespeichert (9). Nach einer Fallunterscheidung, die überprüft, ob das Kopiergerät sich im Zustand `EXPRESPAY` befindet (10), wird der Zustand des Terminals auf `TERMINAL_IDLE` zurückgesetzt (11). Im Anschluss daran wird überprüft, ob der in der Nachricht enthaltene und in der lokalen Variablen `authhashed` gespeicherte Hashwert korrekt ist (12). Hierfür wird zunächst ein `AuthData`-Objekt mit den richtigen Daten erzeugt (`new AuthData(Constants.PAY, passphrase, challenge, amountToPay)`). Über diesen Daten muss nun der Hashwert gebildet werden. Hierfür wird die statische `hash`-Methode der Klasse `HashedData` verwendet. Nun müssen das Objekt `authhashed` und das neu

erzeugte `HashedData`-Objekt verglichen werden. Da der ==-Operator in Java nur Objektgleichheit und keine Wertgleichheit überprüft, wird hierfür die von jeder Klasse implementierte `equals`-Methode verwendet. Ist der Hashwert korrekt, wird abschließend eine `UIssueCopies`-Nachricht versendet, diese enthält den bezahlten Betrag (13).

A.2.3. Deploymentdiagramm

Das Deploymentdiagramm des Terminal-PSMs entspricht dem Diagramm des PIMs. Die beiden möglichen Deploymentdiagramme der Kopierkartenanwendung sind in Abbildung A.13 dargestellt.

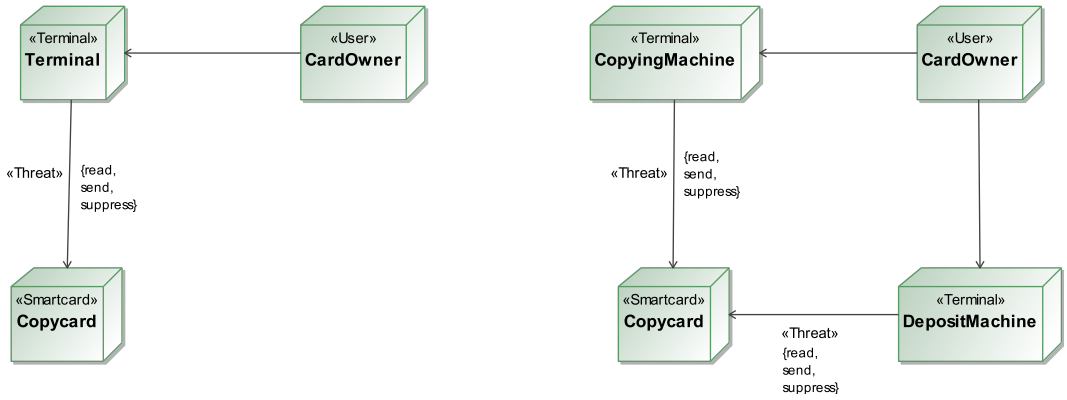

Abbildung A.13.: Zwei mögliche Deploymentdiagramme für die Kopierkartenanwendung

A.3. Beispiel Kopierkartenanwendung: Das Smart Card-PSM

In diesem Abschnitt wird das plattformspezifische Modell für die Smart Card am Beispiel der Kopierkarte illustriert. Bei Übereinstimmungen mit dem Terminal-PSM werden diese nicht nochmal ausführlich dargestellt, sondern nur auf die Unterschiede eingegangen.

A.3.1. Klassendiagramme

Abbildung A.14 zeigt die Komponentenklasse `Copycard` mit ihren Attributen und der Assoziation zur Zustandsklasse.

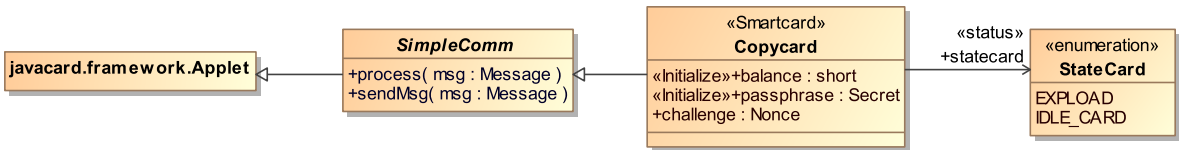

Abbildung A.14.: Ausschnitt aus dem plattformspezifischen Smart Card-PSM: Klassen für die Smart Card und assoziierte Klassen

Einer besseren Strukturierung wegen sind die Methoden, die für die Kommunikation mit einem Terminal von Bedeutung sind, in der abstrakten Klasse `SimpleComm` ausgelagert. Diese ist eine Oberklasse der Klasse `Copycard` und besitzt die Methoden `process(msg : Message)` zum Verarbeiten einer empfangenen Nachricht und `sendMsg(msg : Message)` zum Versenden einer Nachricht. Die Klasse `Copycard` stellt eine Besonderheit dar, weil sie die Appletklasse repräsentiert, die später auf die Smart Card geladen und dort ausgeführt werden soll. Diese Klasse muss von der Klasse `javacard.framework.Applet` abgeleitet sein (siehe [90]). Die Klasse `SimpleComm` ist deshalb eine Subklasse von `javacard.framework.Applet`.

Die Klasse `Copycard` besitzt dieselben Attribute wie im PIM. Die primitiven Datentypen der Attribute wurden jedoch durch Java Card-spezifische Datentypen ersetzt. Somit haben die Attribute `balance` und `passphrase` den Typ `short`, `challenge` hat den Typ `Nonce`. Die möglichen Zustände der Kopierkarte sind in der Klasse `StateCard` modelliert. Die Klasse `Copycard` besitzt keinen Konstruktor, da die Initialisierung eines Applets durch Senden einer entsprechenden Initialisierungsnachricht an die Smart Card erfolgt.

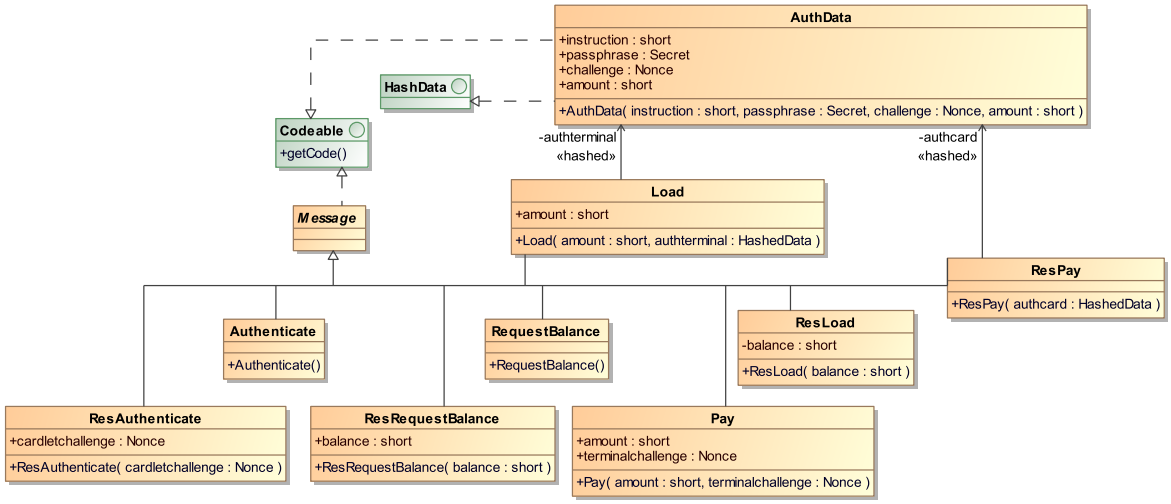

Abbildung A.15.: Ausschnitt aus dem Smart Card-PSM: Klassen für die Nachrichten und assoziierte Klassen

Abbildung A.15 zeigt die Nachrichtenklassen sowie deren assoziierte Klassen des Smart Card-PSMs. Die Klassen sind sehr ähnlich zu denen des Terminals-PSMs. Die primitiven Datentypen sind nun Java Card-spezifisch. Ein weiterer Unterschied ist, dass die Smart Card nur mit dem Terminal und nicht direkt mit einem Benutzer kommuniziert. Aus diesem Grund sind die `Usermessages` im Smart Card-PSM nicht vorhanden.

In Abbildung A.16 sind die Klassen `Code`, `Coding` und `Constants` abgebildet. Die Klassen `Coding` und `Code` sind für die Serialisierung und Deserialisierung der Daten notwendig. Die Klasse `Constants` speichert die vom Entwickler modellierten Konstanten. Da die Konstanten `PAY` und `LOAD` im PIM keine Typangabe und keinen Wert besitzen, bekommen sie im Smart Card-PSM den Typ `short` und werden von eins an aufsteigend nummeriert.

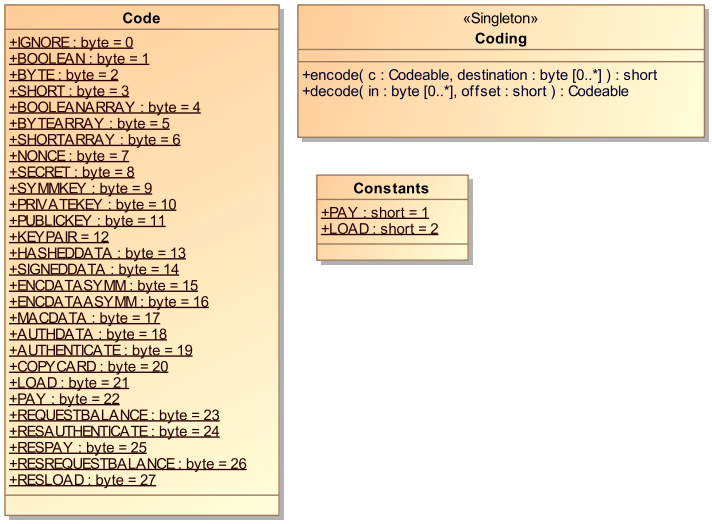

Abbildung A.16.: Ausschnitt aus dem Smart Card-PSM: Klassen `Code`, `Coding` und `Constants`

Die Klasse `Store` ist in Abbildung A.17 abgebildet. Diese Klasse enthält den Objektmanager für die Smart Card-Komponente `Copycard`.

Abbildung A.17.: Ausschnitt aus dem Smart Card-PSM: Die Klasse `Store` für die Speicherverwaltung

Der `Store` enthält Methoden für die Herausgabe der verwalteten Objekte. Für die Klasse `AuthData` beispielsweise gibt es die Methode `newAuthData()`, die ein verwaltetes Ob-

jekt vom Typ `AuthData` zurückgibt. Die Attribute und Assoziationen des Objekts sind mit Defaultwerten vorinitialisiert. Bei Aufruf der Methode mit Argumenten liefert diese ein Objekt vom Typ `AuthData` zurück, bei dem die Attribute entsprechend der übermittelten Werte gesetzt wurden.

A.3.2. Aktivitätsdiagramme

Im Folgenden sind die Aktivitätsdiagramme des plattformspezifischen Smart Card-Modells beschrieben. Wie im Terminal-PSM enthält ein Aktivitätsdiagramm immer genau einen Aktivitätsbereich. Dies ist im Smart Card-PSM der Aktivitätsbereich für die Kopierkartenkomponente. Die im PIM enthaltenen MEL-Ausdrücke sind im Smart Card-PSM durch entsprechende Java Card-Ausdrücke ersetzt worden.

Abfragen des Kontostands:
Abbildung A.18 zeigt das Aktivitätsdiagramm des Smart Card-PSMs zum Abfragen des Kontostands.

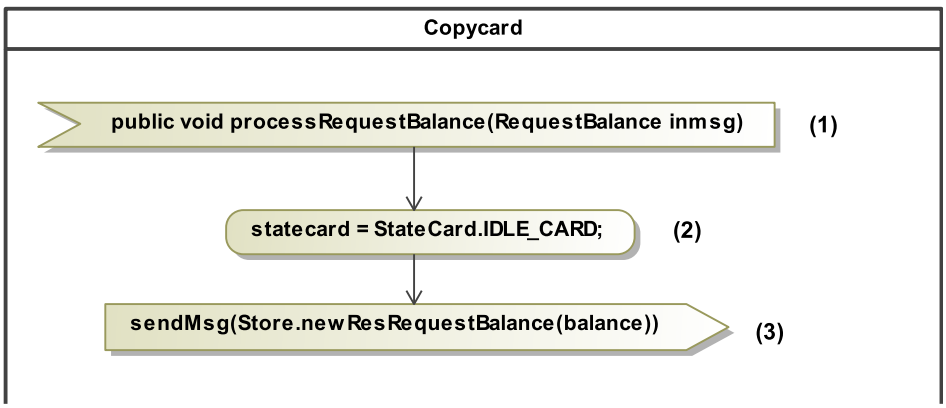

Abbildung A.18.: Ausschnitt aus dem Smart Card-PSM: Aktivitätsdiagramm für das Abfragen des Kontostandes

Das Protokoll zum Abfragen des Kontostands besteht für die Karte aus nur einem Protokollschritt. Dieser beginnt für die Komponente `Copycard` mit dem Empfangen einer `RequestBalance`-Nachricht. Dies ist in einer `AcceptEventAction` durch den Aufruf der Methode `processRequestBalance` modelliert (1). Anschließend wird der Zustand `statecard` der Kopierkarte auf `IDLE_CARD` gesetzt (2). Anders als im Terminal-PSM wird auf die Attribute und Assoziationsenden der Kartenkomponenten nicht über `get-` und `set-`Methoden zugegriffen. Abschließend versendet die Kopierkarte eine `ResRequestBalance`-Nachricht. Dies ist in einer `SendSignalAction` durch Aufruf der `sendMsg`-Methode modelliert. Das benötigte Nachrichtenobjekt wird nicht neu erzeugt, sondern vom Objektmanager zur Verfügung gestellt. Dies geschieht durch Aufruf der Methode `newRequestBalance` in der Klasse `Store`.

Aufladen der Kopierkarte:
Das Aktivitätsdiagramm zum Aufladen der Kopierkarte ist in Abbildung A.19 abgebildet. Das Protokoll besteht auf Seite der Kopierkarte aus zwei Schritten.

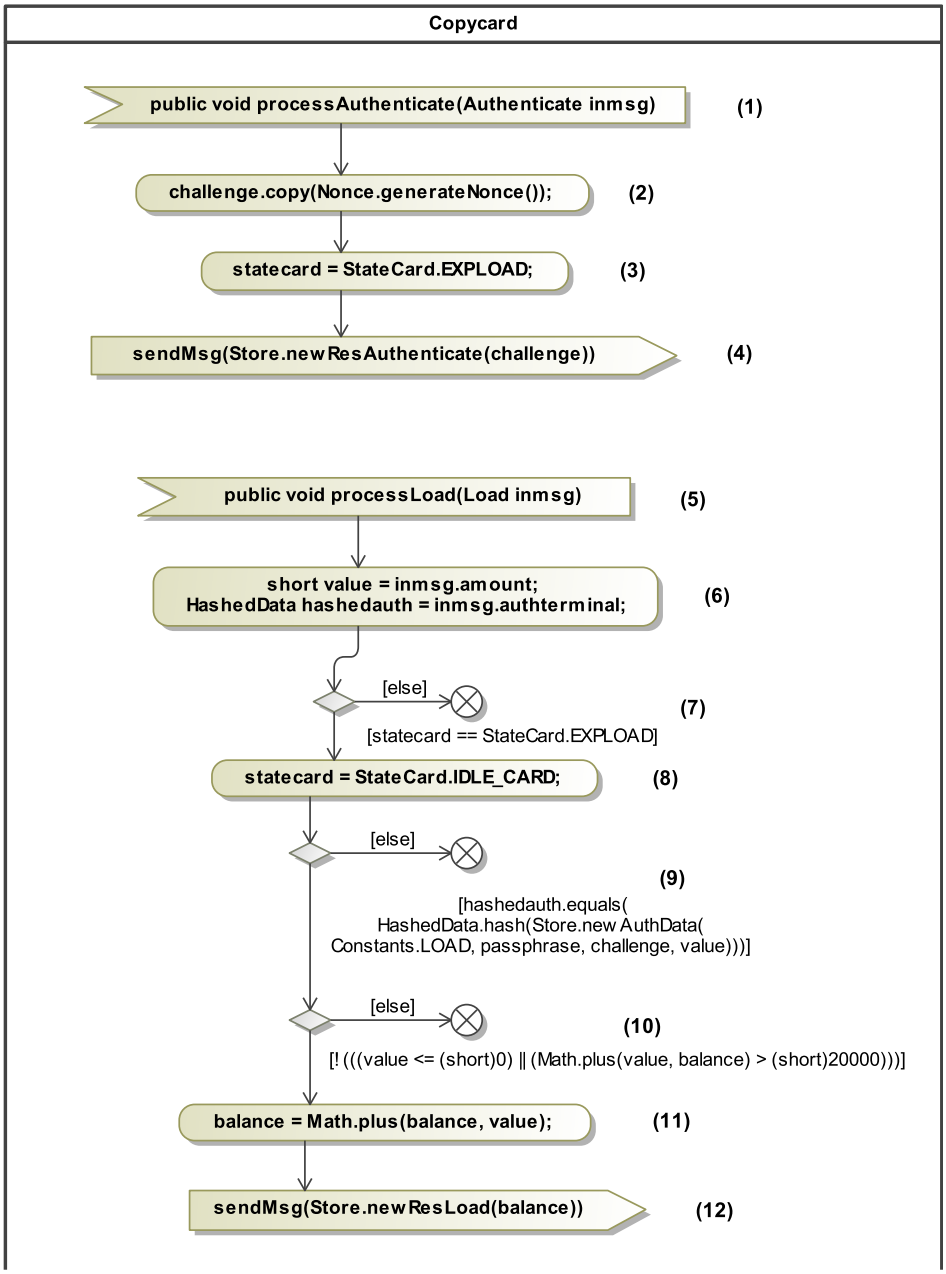

Abbildung A.19.: Ausschnitt aus dem Smart Card-PSM: Aktivitätsdiagramm für das Aufladen der Karte

Der erste Protokollschritt beginnt mit dem Empfangen einer `Authenticate`-Nachricht bzw. dem Aufruf der `processAuthenticate`-Methode (1). Im nächsten Schritt wird durch Aufruf der statischen Methode `generateNonce()` eine neue Nonce erzeugt und dem Attribut `challenge` der Kopierkarte zugewiesen. Bei Zuweisungen an Attribute (nichtprimitiven Typs) und Assoziationsenden der Komponentenklassen wird das zugewiesene Objekt immer kopiert anstatt nur die Referenz auf dieses Objekt zu speichern (siehe Abschnitt 10.3.4). Aus

diesem Grund wird für das Attribut challenge die copy-Methode aufgerufen, die eine Kopie der neu erzeugten Nonce im Attribut challenge speichert (2). Anschließend wird der Zustand der Kopierkarte auf EXPLOAD gesetzt (3) und eine ResAuthenticate-Nachricht gesendet. Das Nachrichtenobjekt wird nicht neu erzeugt, sondern ein entsprechendes Objekt vom Objektmanager zur Verfügung gestellt (4).

Der zweite Protokollschritt beginnt mit dem Empfangen einer Load-Nachricht (5). Die in der Nachricht enthaltenen Daten werden in den lokalen Variablen value und hashedauth zwischengespeichert. Der Ausdruck inmsg.amount selektiert den im Attribut amount der Nachricht gespeicherten Wert, inmsg.authterminal den in der Nachricht enthaltenen Hashwert. amount und authterminal sind die Attribute der Klasse Load (6). Im Anschluss erfolgt die Abfrage, ob der Zustand der Kopierkarte gleich EXPLOAD ist (7), anschließend wird der Zustand auf IDLE_CARD gesetzt (8). Der nächste Schritt ist das Überprüfen der Hashwerte. Hierfür wird der in hashedauth gespeicherte Hashwert mit einem über einem AuthData-Objekt erzeugten Hashwert verglichen. Das benötigte AuthData-Objekt wird vom Objektmanager zur Verfügung gestellt. Zum Bilden des Hashwertes wird die statische Methode hash verwendet. Der Vergleich der Werte geschieht durch Aufruf der equals-Methode, da ein Hashwert eine Folge von Bytes, gespeichert in einem Byte-Array, ist (9). Im nächsten Schritt wird geprüft, ob der abzubuchende Betrag value größer als null und die Summe des jetzigen Kontostands und des zu ladenden Betrags kleiner gleich 20.000 ist (10). Die Addition wird in den Aufruf der plus-Methode übersetzt. Diese prüft zunächst, ob die Addition zu einem Über- oder Unterlauf (des short-Wertebereichs) führt. In diesem Fall wird eine Exception geworfen und die Addition nicht durchgeführt. Anderenfalls führt die Methode die Addition durch. Anschließend wird der Kontostand um den in value gespeicherten Wert erhöht (11) und eine ResLoad-Nachricht gesendet (12).

Anfertigen von Kopien:
In Abbildung A.20 ist das Aktivitätsdiagramm zum Anfertigen von Kopien dargestellt.

Das Protokoll beginnt auf Seite der Kopierkarte mit dem Empfangen einer Pay-Nachricht (1). Die in der Nachricht enthaltenen Daten werden in den lokalen Variablen value und termchallenge gespeichert (2). Anschließend wird der Zustand der Kopierkarte auf IDLE_CARD gesetzt. Daraufhin wird geprüft, ob der abzubuchende Betrag value größer als null ist und noch genug Geld auf der Karte gespeichert ist, um den Betrag value abzubuchen (4). Ist dies der Fall, wird der Kontostand balance um den Wert value reduziert. Wie auch im Load-Protokoll ist an dieser Stelle sichtbar, dass die arithmetischen Operationen auf Smart Card-Seite in Methodenaufrufe übersetzt werden. Die zugehörigen Methoden werden automatisch generiert und prüfen, ob bei Durchführung der Operation ein Over- oder Underflow auftritt. Es wird davon ausgegangen, dass der Modellierer dies nicht wünscht und es wird eine Exception geworfen. Sonst wird die Operation durchgeführt (5). Anschließend wird ein neues AuthData-Objekt vom Objektmanager zur Verfügung gestellt (6) und über diesem der Hashwert gebildet und in der lokalen Variablen authhashed gespeichert (7). Abschließend wird eine ResPay-Nachricht aus dem Objektmanager angefordert und versendet. Diese enthält den zuvor erzeugten Hashwert (8).

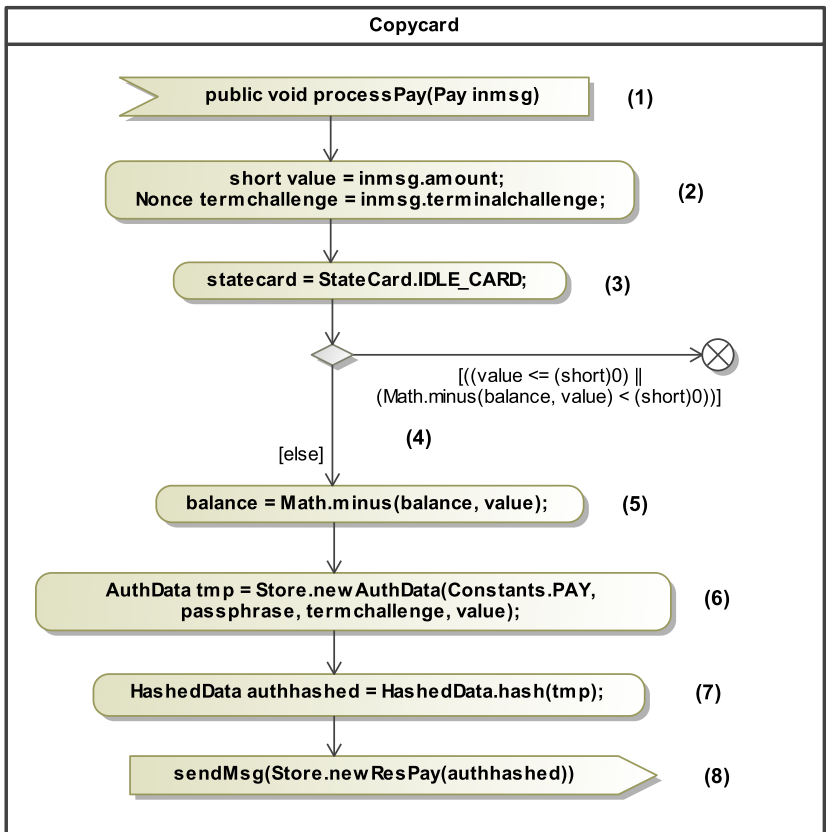

Abbildung A.20.: Ausschnitt aus dem Smart Card-PSM: Aktivitätsdiagramm für das Anfertigen von Kopien

A.3.3. Deploymentdiagramm

Das Deploymentdiagramm des Smart Card-PSMs entspricht dem Deploymentdiagramm des PIMs (siehe Abschnitt 4.2.2).

B Quellcode der Kopierkartenanwendung

Zusammenfassung: Dieses Kapitel enthält den generierten Quellcode der Kopierkartenanwendung. Er lässt sich in drei Teile, einen für jeden Komponententyp, unterteilen: Den Code für die Ladegeräte, den Code für die Kopiergeräte sowie den Code für die Kopierkarten. Für jedes dieser Teile wird ein eigenes Java-Paket generiert, das den Code der jeweiligen Komponente enthält. Der Code der Anwendung ist auch auf der Projektwebseite[1] zu finden.

Abschnitt B.1 enthält den Code, der auf die Kopierkarten geladen wird. In Abschnitt B.2 ist der generierte Code für die Terminals dargestellt. Da der Code der Kopiergeräten an vielen Stellen ähnlich zu dem der Ladegeräte ist, wird im Folgenden nur der Quellcode für die Ladeterminals abgebildet.

B.1. Smart Card-Code der Kopierkartenanwendung

Dieser Abschnitt enthält den generierten Java Card-Code für die Kopierkarten.

B.1.1. Daten- und Nachrichtenklassen

Im Folgenden sind die generierte Java Card-Klassen `AuthData`, `Authenticate`, `Load`, `Message`, `Pay`, `RequestBalance`, `ResAuthenticate`, `ResLoad`, `ResPay` und `ResRequestBalance` abgebildet.

```
public class AuthData implements HashData, Codeable {
public short instruction;
public Secret passphrase;
public Nonce challenge;
public short amount;

public AuthData() {
   instruction = 0;
   passphrase = new Secret();
```

[1] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/

```
   challenge = new Nonce();
   amount = 0;}

public AuthData(short instruction, Secret passphrase,
      Nonce challenge, short amount) {
   this.instruction = instruction;
   this.passphrase = passphrase;
   this.challenge = challenge;
   this.amount = amount;}

public byte getCode() {
   return Code.AUTHDATA;}

public boolean equals(AuthData other) {
   if (this.instruction != other.instruction)
      return false;
   if (!this.passphrase.equals(other.passphrase))
      return false;
   if (!this.challenge.equals(other.challenge))
      return false;
   if (this.amount != other.amount)
      return false;
   return true;}

public void copy(AuthData from) {
   this.instruction = from.instruction;
   this.passphrase.copy(from.passphrase);
   this.challenge.copy(from.challenge);
   this.amount = from.amount;}
}
```

Listing B.1: Smart Card-Code der Klasse `AuthData`

```
public class Authenticate extends Message {
public Authenticate() {}

public byte getCode() {
   return Code.AUTHENTICATE;}

public boolean equals(Authenticate other) {
   return true;}

public void copy(Authenticate from) {}
}
```

Listing B.2: Smart Card-Code der Klasse `Authenticate`

```
public class Load extends Message {
public short amount;
```

```
public HashedData authterminal;
public Load() {
   amount = 0;
   authterminal = new HashedData();}

public Load(short amount, HashedData authterminal) {
   this.amount = amount;
   this.authterminal = authterminal;}

public byte getCode() {
   return Code.LOAD;}

public boolean equals(Load other) {
   if (this.amount != other.amount)
      return false;
   if (!this.authterminal.equals(other.authterminal))
      return false;
   return true;}

public void copy(Load from) {
   this.amount = from.amount;
   this.authterminal.copy(from.authterminal);}
}
```

Listing B.3: Smart Card-Code der Klasse `Load`

```
public abstract class Message implements Codeable {
public Message() {}

public byte getCode() {
   return Code.MESSAGE;}

public boolean equals(Message other) {
      return true;}}
```

Listing B.4: Smart Card-Code der Klasse `Message`

```
public class Pay extends Message {
public short amount;
public Nonce terminalchallenge;
public Pay() {
   amount = 0;
   terminalchallenge = new Nonce();}

public Pay(short amount, Nonce terminalchallenge) {
   this.amount = amount;
   this.terminalchallenge = terminalchallenge;}

public byte getCode() {
```

```
   return Code.PAY;}

public boolean equals(Pay other) {
   if (this.amount != other.amount)
      return false;
   if (!this.terminalchallenge.equals(other.terminalchallenge))
      return false;
   return true;}

public void copy(Pay from) {
   this.amount = from.amount;
   this.terminalchallenge.copy(from.terminalchallenge);}}
```

Listing B.5: Smart Card-Code der Klasse `Pay`

```
public class RequestBalance extends Message {
public RequestBalance() {}

public byte getCode() {
   return Code.REQUESTBALANCE;}

public boolean equals(RequestBalance other) {
   return true;}

public void copy(RequestBalance from) {}}
```

Listing B.6: Smart Card-Code der Klasse `RequestBalance`

```
public class ResAuthenticate extends Message {
public Nonce cardletchallenge;
public ResAuthenticate() {
   cardletchallenge = new Nonce();}

public ResAuthenticate(Nonce cardletchallenge) {
      this.cardletchallenge = cardletchallenge;}

public byte getCode() {
   return Code.RESAUTHENTICATE;}

public boolean equals(ResAuthenticate other) {
   if (!this.cardletchallenge.equals(other.cardletchallenge))
      return false;
   return true;}

public void copy(ResAuthenticate from) {
   this.cardletchallenge.copy(from.cardletchallenge);}}
```

Listing B.7: Smart Card-Code der Klasse `ResAuthenticate`

```
public class ResLoad extends Message {
```

```
public short balance;
public ResLoad() {
   balance = 0;}

public ResLoad(short balance) {
   this.balance = balance;}

public byte getCode() {
   return Code.RESLOAD;}

public boolean equals(ResLoad other) {
   if (this.balance != other.balance)
      return false;
   return true;}

public void copy(ResLoad from) {
   this.balance = from.balance;}
}
```

Listing B.8: Smart Card-Code der Klasse `ResLoad`

```
public class ResPay extends Message {
public HashedData authcard;
public ResPay() {
   authcard = new HashedData();}

public ResPay(HashedData authcard) {
   this.authcard = authcard;}

public byte getCode() {
   return Code.RESPAY;}

public boolean equals(ResPay other) {
   if (!this.authcard.equals(other.authcard))
      return false;
   return true;}

public void copy(ResPay from) {
   this.authcard.copy(from.authcard);}
}
```

Listing B.9: Smart Card-Code der Klasse `ResPay`

```
public class ResRequestBalance extends Message {
public short balance;
public ResRequestBalance() {
   balance = 0;}

public ResRequestBalance(short balance) {
```

```
   this.balance = balance;}

public byte getCode() {
   return Code.RESREQUESTBALANCE;}

public boolean equals(ResRequestBalance other) {
   if (this.balance != other.balance)
      return false;
   return true;}

public void copy(ResRequestBalance from) {
   this.balance = from.balance;}
}
```

Listing B.10: Smart Card-Code der Klasse `ResRequestBalance`

B.1.2. Klassen für die (De-)Serialisierung

```
public interface Codeable {
   byte getCode();}
```

Listing B.11: Smart Card-Code des Interfaces `Codeable`

```
public class Code{
   public final static byte IGNORE = (byte) 0;
   public final static byte NULL = (byte) 1;
   public final static byte BOOLEAN = (byte) 2;
   public final static byte BYTE = (byte) 3;
   public final static byte INT = (byte) 4;
   public final static byte BOOLEANARRAY = (byte) 5;
   public final static byte BYTEARRAY = (byte) 6;
   public final static byte SHORTARRAY = (byte) 7;
   public final static byte NONCE = (byte) 8;
   public final static byte SECRET = (byte) 9;
   public final static byte SYMMKEY = (byte) 10;
   public final static byte PRIVATEKEY = (byte) 11;
   public final static byte PUBLICKEY = (byte) 12;
   public final static byte HASHEDDATA = (byte) 13;
   public final static byte SIGNEDDATA = (byte) 14;
   public final static byte ENCDATASYMM = (byte) 15;
   public final static byte ENCDATAASYMM = (byte) 16;
   public final static byte MACDATA = (byte) 17;
   public final static byte AUTHDATA = (byte) 18;
   public final static byte AUTHENTICATE = (byte) 19;
   public final static byte COPYCARD = (byte) 20;
   public final static byte LOAD = (byte) 21;
   public final static byte MESSAGE = (byte) 22;
   public final static byte PAY = (byte) 23;
```

```
   public final static byte REQUESTBALANCE = (byte) 24;
   public final static byte RESAUTHENTICATE = (byte) 25;
   public final static byte RESLOAD = (byte) 26;
   public final static byte RESPAY = (byte) 27;
   public final static byte RESREQUESTBALANCE = (byte) 28;
}
```

Listing B.12: Smart Card-Code der Klasse `Code`

```
public class Coding {
private short encoding_length;
private byte[] encodeDestination;
private static Coding the_instance;
public static Coding getInstance() {
   if (the_instance == null)
      the_instance = new Coding();
   return the_instance;}

public short getEncodingLength() {
   return encoding_length;}

public void encodeInit(byte[] dest) {
   encodeDestination = dest;
   encoding_length = 0;}

public void decodeInit() {
   encoding_length = 0;}

private byte decodeByte(byte[] in) {
   if (in[encoding_length++] != Code.BYTE) {
      SimpleComm.stop();}

   byte res = in[encoding_length++];
   return res;
}

private boolean decodeBoolean(byte[] in) {
   if (in[encoding_length++] != Code.BOOLEAN) {
      SimpleComm.stop();}
   boolean res = (in[encoding_length++] == 1 ? true : false);
   return res;   }

public short encode(Codeable o, byte[] dest) {
   encodeInit(dest);
   encode(o);
   return encoding_length;}

public short computeSize(Codeable o) {
```

```
   encoding_length = 0;
   try {
      compSize(o);
      return encoding_length;}
   catch (Exception e) {
      return -1;}
}

private void encodeInt(short o) {
   encodeDestination[encoding_length++] = Code.INT;
   Util.setShort(encodeDestination, encoding_length,
            (short) 0x00);
   Util.setShort(encodeDestination,
            (short) (encoding_length + 2), o);
   encoding_length += 4;}

public void encodeByte(byte o) {
   encodeDestination[encoding_length++] = Code.BYTE;
   encodeDestination[encoding_length++] = o;}

private void encodeBoolean(boolean o) {
   encodeDestination[encoding_length++] = Code.BOOLEAN;
   encodeDestination[encoding_length++] = (byte) (o ? 1 : 0);}

private void compSizeInt(short o) {
   encoding_length += 5;}

private void compSizeByte(byte o) {
   encoding_length += 2;}

private void compSizeBoolean(boolean o) {
   encoding_length += 2;}

private short decodeInt(byte[] in) {
   if (in[encoding_length++] != Code.INT)
      SimpleComm.stop();
   short upper = Util.getShort(in, encoding_length);
   if (upper != 0x00) {
      SimpleComm.stop();
      return 0;}
   else {
      short lower = Util.getShort(in,
                           (short) (encoding_length + 2));
         encoding_length += 4;
      return lower;}
}
```

```
public void encodeByteArray(byte[] o) {
   encodeDestination[encoding_length++] = Code.BYTEARRAY;
   Util.setShort(encodeDestination, encoding_length,
                 (short) (o.length));
   encoding_length += 2;
   Util.arrayCopy(o, (short) 0, encodeDestination,
                  encoding_length, (short) o.length);
   encoding_length += o.length;
}

private void compSizeByteArray(byte[] o) {
   encoding_length += 3;
   encoding_length += o.length;
}

private void decodeByteArrayInto(byte[] in, byte[] into) {
   if (in[encoding_length++] != Code.BYTEARRAY)
      SimpleComm.stop();
   short len = Util.getShort(in, encoding_length);
   encoding_length += 2;
   if (!(0 <= len && len <= into.length
         && len <= (in.length - encoding_length)))
      SimpleComm.stop();
   Util.arrayCopy(in, encoding_length, into, (short) 0, len);
   if (len < into.length)
      Util.arrayFillNonAtomic(into, len,
            ((short) (into.length - len)), (byte) 0);
   encoding_length += len;
}

public void encode(Codeable c) {
   switch (c.getCode()) {
      case Code.LOAD :
         encodeLoad((Load) c);
         break;
      case Code.REQUESTBALANCE :
         encodeRequestBalance((RequestBalance) c);
         break;
      case Code.PAY :
         encodePay((Pay) c);
         break;
      case Code.RESREQUESTBALANCE :
         encodeResRequestBalance((ResRequestBalance) c);
         break;
      case Code.RESPAY :
         encodeResPay((ResPay) c);
         break;
```

```
      case Code.RESAUTHENTICATE :
         encodeResAuthenticate((ResAuthenticate) c);
         break;
      case Code.AUTHENTICATE :
         encodeAuthenticate((Authenticate) c);
         break;
      case Code.RESLOAD :
         encodeResLoad((ResLoad) c);
         break;
      case Code.HASHEDDATA :
         encodeHashedData((HashedData) c);
         break;
      case Code.AUTHDATA :
         encodeAuthData((AuthData) c);
         break;
      case Code.NONCE :
         encodeNonce((Nonce) c);
         break;
      case Code.SECRET :
         encodeSecret((Secret) c);
         break;
      default : SimpleComm.stop();
      }
}

public Message decodeMessage(byte[] in, short offset,
                             short expectedLength) {
   encoding_length = offset;
   Message m = null;
   try {
      switch (in[encoding_length]) {
         case Code.LOAD :
            m = Store.newLoad();
            decodeLoadInto(in, (Load) m);
            break;
         case Code.REQUESTBALANCE :
            m = Store.newRequestBalance();
            decodeRequestBalanceInto(in, (RequestBalance) m);
            break;
         case Code.PAY :
            m = Store.newPay();
            decodePayInto(in, (Pay) m);
            break;
         case Code.RESREQUESTBALANCE :
            m = Store.newResRequestBalance();
            decodeResRequestBalanceInto(in,
                                 (ResRequestBalance) m);
```

```
            break;
         case Code.RESPAY :
            m = Store.newResPay();
            decodeResPayInto(in, (ResPay) m);
            break;
         case Code.RESAUTHENTICATE :
            m = Store.newResAuthenticate();
            decodeResAuthenticateInto(in, (ResAuthenticate) m);
            break;
         case Code.AUTHENTICATE :
            m = Store.newAuthenticate();
            decodeAuthenticateInto(in, (Authenticate) m);
            break;
         case Code.RESLOAD :
            m = Store.newResLoad();
            decodeResLoadInto(in, (ResLoad) m);
            break;
      }
   } catch (ArrayIndexOutOfBoundsException e) {
      SimpleComm.stop();
   }
   if (m == null
       || encoding_length != (short) (expectedLength + offset))
      SimpleComm.stop();
   return m;
}

public void compSize(Codeable c) {
   switch (c.getCode()) {
      case Code.LOAD :
               compSizeLoad((Load) c);
               break;
      case Code.REQUESTBALANCE :
               compSizeRequestBalance((RequestBalance) c);
               break;
      case Code.PAY :
               compSizePay((Pay) c);
               break;
      case Code.RESREQUESTBALANCE :
               compSizeResRequestBalance((ResRequestBalance) c);
               break;
      case Code.RESPAY :
               compSizeResPay((ResPay) c);
               break;
      case Code.RESAUTHENTICATE :
               compSizeResAuthenticate((ResAuthenticate) c);
               break;
```

```
      case Code.AUTHENTICATE :
              compSizeAuthenticate((Authenticate) c);
              break;
      case Code.RESLOAD :
              compSizeResLoad((ResLoad) c);
              break;
      case Code.HASHEDDATA :
              compSizeHashedData((HashedData) c);
              break;
      case Code.AUTHDATA :
              compSizeAuthData((AuthData) c);
              break;
      case Code.NONCE :
              compSizeNonce((Nonce) c);
              break;
      case Code.SECRET :
              compSizeSecret((Secret) c);
              break;
      default :
              SimpleComm.stop();
              break;}
}

private void encodeLoad(Load c) {
   encodeDestination[encoding_length++] = Code.LOAD;
   encodeInt((short) c.amount);
   encode(c.authterminal);}

private void decodeLoadInto(byte[] in, Load into) {
   if (in[encoding_length++] != Code.LOAD)
      SimpleComm.stop();
   into.amount = decodeInt(in);
   decodeHashedDataInto(in, (HashedData) (into.authterminal));}

private void compSizeLoad(Load c) {
   encoding_length++;
   compSizeInt(c.amount);
   compSize(c.authterminal);}

private void encodeRequestBalance(RequestBalance c) {
   encodeDestination[encoding_length++] = Code.REQUESTBALANCE;}

private void decodeRequestBalanceInto(byte[] in,
                  RequestBalance into) {
   if (in[encoding_length++] != Code.REQUESTBALANCE)
      SimpleComm.stop();}
```

```
private void compSizeRequestBalance(RequestBalance c) {
   encoding_length++;}

private void encodePay(Pay c) {
   encodeDestination[encoding_length++] = Code.PAY;
   encodeInt((short) c.amount);
   encode(c.terminalchallenge);}

private void decodePayInto(byte[] in, Pay into) {
   if (in[encoding_length++] != Code.PAY)
      SimpleComm.stop();
   into.amount = decodeInt(in);
   decodeNonceInto(in, (Nonce) (into.terminalchallenge));}

private void compSizePay(Pay c) {
   encoding_length++;
   compSizeInt(c.amount);
   compSize(c.terminalchallenge);}

private void encodeResRequestBalance(ResRequestBalance c) {
   encodeDestination[encoding_length++]
        = Code.RESREQUESTBALANCE;
   encodeInt((short) c.balance);}

private void decodeResRequestBalanceInto(byte[] in,
                ResRequestBalance into) {
   if (in[encoding_length++] != Code.RESREQUESTBALANCE)
      SimpleComm.stop();
   into.balance = decodeInt(in);}

private void compSizeResRequestBalance(ResRequestBalance c) {
   encoding_length++;
   compSizeInt(c.balance);}

private void encodeResPay(ResPay c) {
   encodeDestination[encoding_length++] = Code.RESPAY;
   encode(c.authcard);}

private void decodeResPayInto(byte[] in, ResPay into) {
   if (in[encoding_length++] != Code.RESPAY)
      SimpleComm.stop();
   decodeHashedDataInto(in, (HashedData) (into.authcard));}

private void compSizeResPay(ResPay c) {
   encoding_length++;
   compSize(c.authcard);}
```

```
private void encodeResAuthenticate(ResAuthenticate c) {
   encodeDestination[encoding_length++] = Code.RESAUTHENTICATE;
   encode(c.cardletchallenge);}

private void decodeResAuthenticateInto(byte[] in,
                                       ResAuthenticate into) {
   if (in[encoding_length++] != Code.RESAUTHENTICATE)
      SimpleComm.stop();
   decodeNonceInto(in, (Nonce) (into.cardletchallenge));}

private void compSizeResAuthenticate(ResAuthenticate c) {
   encoding_length++;
   compSize(c.cardletchallenge);}

private void encodeAuthenticate(Authenticate c) {
   encodeDestination[encoding_length++] = Code.AUTHENTICATE;}

private void decodeAuthenticateInto(byte[] in,
                                    Authenticate into) {
   if (in[encoding_length++] != Code.AUTHENTICATE)
      SimpleComm.stop();}

private void compSizeAuthenticate(Authenticate c) {
   encoding_length++;}

private void encodeResLoad(ResLoad c) {
   encodeDestination[encoding_length++] = Code.RESLOAD;
   encodeInt((short) c.balance);}

private void decodeResLoadInto(byte[] in, ResLoad into) {
   if (in[encoding_length++] != Code.RESLOAD)
      SimpleComm.stop();
   into.balance = decodeInt(in);}

private void compSizeResLoad(ResLoad c) {
   encoding_length++;
   compSizeInt(c.balance);}

private void encodeHashedData(HashedData c) {
   encodeDestination[encoding_length++] = Code.HASHEDDATA;
   encodeByteArray((byte[]) c.hashed);}

private void decodeHashedDataInto(byte[] in, HashedData into) {
   if (in[encoding_length++] != Code.HASHEDDATA)
      SimpleComm.stop();
   decodeByteArrayInto(in, into.hashed);}
```

```
private void compSizeHashedData(HashedData c) {
   encoding_length++;
   compSizeByteArray(c.hashed);}

private void encodeAuthData(AuthData c) {
   encodeDestination[encoding_length++] = Code.AUTHDATA;
   encodeInt((short) c.instruction);
   encode(c.passphrase);
   encode(c.challenge);
   encodeInt((short) c.amount);}

private void decodeAuthDataInto(byte[] in, AuthData into) {
   if (in[encoding_length++] != Code.AUTHDATA)
      SimpleComm.stop();
   into.instruction = decodeInt(in);
   decodeSecretInto(in, (Secret) (into.passphrase));
   decodeNonceInto(in, (Nonce) (into.challenge));
   into.amount = decodeInt(in);}

private void compSizeAuthData(AuthData c) {
   encoding_length++;
   compSizeInt(c.instruction);
   compSize(c.passphrase);
   compSize(c.challenge);
   compSizeInt(c.amount);}

private void encodeNonce(Nonce c) {
   encodeDestination[encoding_length++] = Code.NONCE;
   encodeByteArray((byte[]) c.nonce);}

private void decodeNonceInto(byte[] in, Nonce into) {
   if (in[encoding_length++] != Code.NONCE)
      SimpleComm.stop();
   decodeByteArrayInto(in, into.nonce);}

private void compSizeNonce(Nonce c) {
   encoding_length++;
   compSizeByteArray(c.nonce);}

private void encodeSecret(Secret c) {
   encodeDestination[encoding_length++] = Code.SECRET;
   encodeByteArray((byte[]) c.secret);}

private void decodeSecretInto(byte[] in, Secret into) {
   if (in[encoding_length++] != Code.SECRET)
      SimpleComm.stop();
   decodeByteArrayInto(in, into.secret);}
```

```
private void compSizeSecret(Secret c) {
   encoding_length++;
   compSizeByteArray(c.secret);   }

public void decodeCopycard(byte[] in, short offset,
                           Copycard thisCopycard) {
   encoding_length = offset;
   try {
      if (in[encoding_length++] != Code.COPYCARD)
         SimpleComm.stop();
      if (in[encoding_length] != Code.IGNORE) {
         thisCopycard.balance = decodeInt(in);}
      else encoding_length++;
      if (in[encoding_length] != Code.IGNORE) {
         decodeSecretInto(in, thisCopycard.passphrase);}
      else encoding_length++;

      if (in[encoding_length] != Code.IGNORE) {
         thisCopycard.statecard = decodeByte(in);}
      else encoding_length++;}
   catch (ArrayIndexOutOfBoundsException e) {
         SimpleComm.stop();}
   }
}
```

Listing B.13: Smart Card-Code der Klasse `Coding`

B.1.3. Komponentenklasse `Copycard` mit den implementierten Protokollschritten der Komponente

```
public abstract class SimpleComm extends Applet {
public final static byte INIT_INSTRUCTION = (byte) 4;
public final static byte RESUME_INSTRUCTION = (byte) 6;
public final static short ENCBUF_MAXLEN = 30;
public final static short DECBUF_MAXLEN = 30;
public final static short APDU_MAXLEN = 255;
public final static short OUTMSG_MAXLEN = 254;

protected static Coding coding;

private Message inmsg;
private static Message outmsg;
protected byte[] encodebuffer;
protected byte[] decodebuffer;
private short len = 0;

private static short encbuf_sentlen = 0;
```

```
private static short decbuf_receivedlen = 0;

public void process(APDU apdu) {
   if (selectingApplet())
      return;
   short expected = apdu.setIncomingAndReceive();
   if (expected == 0)
      ISOException.throwIt(
            ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   byte ins = apdu.getBuffer()[ISO7816.OFFSET_INS];

   if (ins == RESUME_INSTRUCTION) {
      sendAPDU(apdu, encodebuffer,
               (short) (len - encbuf_sentlen), false);
      return;}

   if ((short) (decbuf_receivedlen + expected)
       > DECBUF_MAXLEN) {
      decbuf_receivedlen = 0;
      ISOException.throwIt(
            ISO7816.SW_CONDITIONS_NOT_SATISFIED);
      return;}

   Util.arrayCopy(apdu.getBuffer(), ISO7816.OFFSET_CDATA,
                  decodebuffer,
         decbuf_receivedlen, expected);
   decbuf_receivedlen += expected;
   if (apdu.getBuffer()[ISO7816.OFFSET_P1] == 1)
      return;

   if (ins == INIT_INSTRUCTION) {
      appletInitialization(decodebuffer, (short) 0);
      decbuf_receivedlen = 0;
      return;}

   Store.reset();
   outmsg = null;
   inmsg = coding.decodeMessage(decodebuffer, (short) 0,
         decbuf_receivedlen);
   decbuf_receivedlen = 0;

   process(inmsg);
   if (outmsg != null) {
      len = coding.encode((Codeable) outmsg, encodebuffer);
      encbuf_sentlen = 0;
      sendAPDU(apdu, encodebuffer, len, true);}
}
```

```
public abstract void process(Message msg);
protected static void sendMsg(Message msg){
   outmsg = msg;}

public void sendAPDU(APDU apdu, byte[] data, short rem_len,
                     boolean first) {
   if (!first && encbuf_sentlen >= len)
      return;
   byte[] buffer = apdu.getBuffer();
   short sendLen = 0;
   if (rem_len > OUTMSG_MAXLEN) {
      buffer[0] = 1;
      Util.arrayCopy(data, encbuf_sentlen, buffer, (short) 1,
            OUTMSG_MAXLEN);
      encbuf_sentlen += OUTMSG_MAXLEN;
      sendLen = APDU_MAXLEN;}
   else {
      buffer[0] = 0;
      Util.arrayCopy(data, encbuf_sentlen, buffer, (short) 1,
                     rem_len);
      encbuf_sentlen = len;
      sendLen = (short) (rem_len + 1);}
   apdu.setOutgoingAndSend((short) 0, sendLen);}

public abstract void appletInitialization(byte[] buf,
                                          short offset);

public static void install(byte[] bArray, short bOffset,
                           byte bLength){
   new Copycard();}

public static void stop(){
   ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);}

public SimpleComm() {
   coding = Coding.getInstance();

   encodebuffer = new byte[ENCBUF_MAXLEN];
   decodebuffer = new byte[DECBUF_MAXLEN];
   Store.initAll();
   register();}}
```

Listing B.14: Smart Card-Code der Klasse `SimpleComm`

```
public class Copycard extends SimpleComm {
public short balance;
public Secret passphrase;
```

```
public Nonce challenge;
public byte statecard;

public Copycard() {
   balance = 0;
   passphrase = new Secret();
   challenge = new Nonce();
   statecard = StateCard.APPLET_UNINITIALIZED;}

public void appletInitialization(byte[] buf, short offset) {
   if (statecard != StateCard.APPLET_UNINITIALIZED)
      stop();
   Coding.getInstance().decodeCopycard(buf, offset, this);}

public Copycard init(short balance, Secret passphrase,
                     byte statecard) {
   this.balance = balance;
   this.passphrase = passphrase;
   this.statecard = statecard;
   challenge = new Nonce();
   return this;}

public Copycard init() {
   balance = 0;
   passphrase = new Secret();
   challenge = new Nonce();
   statecard = StateCard.IDLE_CARD;
   return this;}

public void process(Message inmsg) {
   if (statecard == StateCard.APPLET_UNINITIALIZED)
      stop();
   switch (inmsg.getCode()) {
   case Code.REQUESTBALANCE :
      processRequestBalance((RequestBalance) inmsg);
      break;
   case Code.PAY :
      processPay((Pay) inmsg);
      break;
   case Code.AUTHENTICATE :
      processAuthenticate((Authenticate) inmsg);
      break;
   case Code.LOAD :
      processLoad((Load) inmsg);
      break;
   default :
      stop();}}
```

```
public void processRequestBalance(RequestBalance inmsg) {
   statecard = StateCard.IDLE_CARD;
   sendMsg(Store.newResRequestBalance(balance));}

public void processPay(Pay inmsg) {
   short value = inmsg.amount;
   Nonce termchallenge = inmsg.terminalchallenge;
   statecard = StateCard.IDLE_CARD;
   if (((value <= (short) 0)
       || (Math.minus(balance, value) < (short) 0))) {
      stop();}
   else {
      balance = Math.minus(balance, value);
      AuthData tmp = Store.newAuthData(Constants.PAY,
            passphrase, termchallenge, value);
      HashedData authhashed = HashedData.hash(tmp);
      sendMsg(Store.newResPay(authhashed));}}

public void processAuthenticate(Authenticate inmsg) {
   challenge.copy(Nonce.generateNonce());
   statecard = StateCard.EXPLOAD;
   sendMsg(Store.newResAuthenticate(challenge));}

public void processLoad(Load inmsg) {
   short value = inmsg.amount;
   HashedData hashedauth = inmsg.authterminal;
   if (statecard == StateCard.EXPLOAD) {
      statecard = StateCard.IDLE_CARD;
      AuthData ad = Store.newAuthData(Constants.LOAD,
            passphrase, challenge, value);
      if (((hashedauth.equals(HashedData.hash(ad))
          && (value > (short) 0)) && (Math
            .plus(value, balance) <= (short) 20000))) {
         balance = Math.plus(balance, value);
         sendMsg(Store.newResLoad(balance));}
      else {stop();}}
   else { stop(); }}}
```

Listing B.15: Smart Card-Code der Klasse `Copycard`

```
public class StateCard {
   public static final byte APPLET_UNINITIALIZED = 1;
   public static final byte EXPLOAD = 2;
   public static final byte IDLE_CARD = 3; }
```

Listing B.16: Smart Card-Code der Klasse `StateCard`

B.1.4. Objektmanager

```
public class Store {
   public final static short LENGTHOFSTRING = 20;
   public final static short LENGTHOFSECRET = 8;
   public final static short LENGTHOFNONCE = 8;
   public final static short COMPUTEDHASHDATALENGTH = 35;
   public final static short HashedDataMaxCountForCopycard = 1;
   public final static short ResLoadMaxCountForCopycard = 1;
   public final static short PayMaxCountForCopycard = 1;
   public final static short LoadMaxCountForCopycard = 1;
   public final static short ResPayMaxCountForCopycard = 1;
   public final static short NonceMaxCountForCopycard = 1;
   public final static short
            ResAuthenticateMaxCountForCopycard = 1;
   public final static short
            RequestBalanceMaxCountForCopycard = 1;
   public final static short
            ResRequestBalanceMaxCountForCopycard = 1;
   public final static short AuthDataMaxCountForCopycard = 1;
   public final static short
            AuthenticateMaxCountForCopycard = 1;

   private static HashedData[] HashedDataArray;
   private static byte HashedDataCount = 0;
   private static void initHashedData() {
      HashedDataArray
        = new HashedData[HashedDataMaxCountForCopycard];
      for (short i = 0; i < HashedDataArray.length; i++)
         HashedDataArray[i] = new HashedData();}

   public static HashedData newHashedData() {
      return HashedDataArray[HashedDataCount++];}

   private static ResLoad[] ResLoadArray;
   private static byte ResLoadCount = 0;
   private static void initResLoad() {
      ResLoadArray = new ResLoad[ResLoadMaxCountForCopycard];
      for (short i = 0; i < ResLoadArray.length; i++)
         ResLoadArray[i] = new ResLoad();}

   public static ResLoad newResLoad() {
      return ResLoadArray[ResLoadCount++];}

   private static Pay[] PayArray;
   private static byte PayCount = 0;
   private static void initPay() {
```

```
      PayArray = new Pay[PayMaxCountForCopycard];
      for (short i = 0; i < PayArray.length; i++)
         PayArray[i] = new Pay();}

   public static Pay newPay() {
      return PayArray[PayCount++];}

   private static Load[] LoadArray;
   private static byte LoadCount = 0;
   private static void initLoad() {
      LoadArray = new Load[LoadMaxCountForCopycard];
      for (short i = 0; i < LoadArray.length; i++)
         LoadArray[i] = new Load();}

   public static Load newLoad() {
      return LoadArray[LoadCount++];}

   private static ResPay[] ResPayArray;
   private static byte ResPayCount = 0;
   private static void initResPay() {
      ResPayArray = new ResPay[ResPayMaxCountForCopycard];
      for (short i = 0; i < ResPayArray.length; i++)
         ResPayArray[i] = new ResPay();}

   public static ResPay newResPay() {
      return ResPayArray[ResPayCount++];}

   private static Nonce[] NonceArray;
   private static byte NonceCount = 0;
   private static void initNonce() {
      NonceArray = new Nonce[NonceMaxCountForCopycard];
      for (short i = 0; i < NonceArray.length; i++)
         NonceArray[i] = new Nonce();}

   public static Nonce newNonce() {
      return NonceArray[NonceCount++];}

   private static ResAuthenticate[] ResAuthenticateArray;
   private static byte ResAuthenticateCount = 0;
   private static void initResAuthenticate() {
      ResAuthenticateArray =
       new ResAuthenticate[ResAuthenticateMaxCountForCopycard];
      for (short i = 0; i < ResAuthenticateArray.length; i++)
         ResAuthenticateArray[i] = new ResAuthenticate();}

   public static ResAuthenticate newResAuthenticate() {
      return ResAuthenticateArray[ResAuthenticateCount++];}
```

```
private static RequestBalance[] RequestBalanceArray;
private static byte RequestBalanceCount = 0;
private static void initRequestBalance() {
   RequestBalanceArray
    = new RequestBalance[RequestBalanceMaxCountForCopycard];
   for (short i = 0; i < RequestBalanceArray.length; i++)
      RequestBalanceArray[i] = new RequestBalance();}

public static RequestBalance newRequestBalance() {
   return RequestBalanceArray[RequestBalanceCount++];}

private static ResRequestBalance[] ResRequestBalanceArray;
private static byte ResRequestBalanceCount = 0;
private static void initResRequestBalance() {
   ResRequestBalanceArray = new
    ResRequestBalance[ResRequestBalanceMaxCountForCopycard];
   for (short i = 0; i < ResRequestBalanceArray.length; i++)
      ResRequestBalanceArray[i] = new ResRequestBalance();}

public static ResRequestBalance newResRequestBalance() {
   return ResRequestBalanceArray[ResRequestBalanceCount++];}

private static AuthData[] AuthDataArray;
private static byte AuthDataCount = 0;
private static void initAuthData() {
   AuthDataArray
     = new AuthData[AuthDataMaxCountForCopycard];
   for (short i = 0; i < AuthDataArray.length; i++)
      AuthDataArray[i] = new AuthData();}

public static AuthData newAuthData() {
   return AuthDataArray[AuthDataCount++];}

private static Authenticate[] AuthenticateArray;
private static byte AuthenticateCount = 0;
private static void initAuthenticate() {
   AuthenticateArray
    = new Authenticate[AuthenticateMaxCountForCopycard];
   for (short i = 0; i < AuthenticateArray.length; i++)
      AuthenticateArray[i] = new Authenticate();}

public static Authenticate newAuthenticate() {
   return AuthenticateArray[AuthenticateCount++];}

public static HashedData newHashedData(byte[] hashed) {
   HashedData _HashedData = newHashedData();
```

```
      Arrays.copy(hashed, _HashedData.hashed);
      return _HashedData;}

   public static ResLoad newResLoad(short balance) {
      ResLoad _ResLoad = newResLoad();
      _ResLoad.balance = balance;
      return _ResLoad;}

   public static Pay newPay(short amount,
            Nonce terminalchallenge) {
      Pay _Pay = newPay();
      _Pay.amount = amount;
      _Pay.terminalchallenge.copy(terminalchallenge);
      return _Pay;}

   public static Load newLoad(short amount,
            HashedData authterminal) {
      Load _Load = newLoad();
      _Load.amount = amount;
      _Load.authterminal.copy(authterminal);
      return _Load;}

   public static ResPay newResPay(HashedData authcard) {
      ResPay _ResPay = newResPay();
      _ResPay.authcard.copy(authcard);
      return _ResPay;}

   public static Nonce newNonce(byte[] nonce) {
      Nonce _Nonce = newNonce();
      Arrays.copy(nonce, _Nonce.nonce);
      return _Nonce;}

   public static ResAuthenticate newResAuthenticate(
            Nonce cardletchallenge) {
      ResAuthenticate _ResAuthenticate = newResAuthenticate();
      _ResAuthenticate.cardletchallenge.copy(cardletchallenge);
      return _ResAuthenticate;}

   public static ResRequestBalance newResRequestBalance(
            short balance) {
      ResRequestBalance _ResRequestBalance
         = newResRequestBalance();
      _ResRequestBalance.balance = balance;
      return _ResRequestBalance;}

   public static AuthData newAuthData(short instruction,
            Secret passphrase, Nonce challenge, short amount) {
```

```
      AuthData _AuthData = newAuthData();
      _AuthData.instruction = instruction;
      _AuthData.passphrase.copy(passphrase);
      _AuthData.challenge.copy(challenge);
      _AuthData.amount = amount;
      return _AuthData;}

   public static void initAll() {
      initHashedData();
      initResLoad();
      initPay();
      initLoad();
      initResPay();
      initNonce();
      initResAuthenticate();
      initRequestBalance();
      initResRequestBalance();
      initAuthData();
      initAuthenticate();}

   public static void reset() {
      HashedDataCount = 0;
      ResLoadCount = 0;
      PayCount = 0;
      LoadCount = 0;
      ResPayCount = 0;
      NonceCount = 0;
      ResAuthenticateCount = 0;
      RequestBalanceCount = 0;
      ResRequestBalanceCount = 0;
      AuthDataCount = 0;
      AuthenticateCount = 0;}
}
```

Listing B.17: Smart Card-Code der Klasse `Store`

B.1.5. Sonstige Klassen

```
public class Constants {
   public static final short PAY = 1;
   public static final short LOAD = 2; }
```

Listing B.18: Smart Card-Code der Klasse `Constants`

```
public class Math {
   public static short plus(short x, short y){
      short res = (short) (x + y);
      if (x < 0 && y < 0 && res >= 0)
         ISOException.throwIt(
```

```
                ISO7816.SW_CONDITIONS_NOT_SATISFIED);
      if (x > 0 && y > 0 && res < 0)
         ISOException.throwIt(
                ISO7816.SW_CONDITIONS_NOT_SATISFIED);
      return res;   }

   public static short minus(short x, short y){
      short res = (short) (x - y);
      if (0 <= x && y < 0 && res < 0)
         ISOException.throwIt(
                ISO7816.SW_CONDITIONS_NOT_SATISFIED);
      if (x < 0 && 0 < y && res > 0)
         ISOException.throwIt(
                ISO7816.SW_CONDITIONS_NOT_SATISFIED);
      return res; }
}
```

Listing B.19: Smart Card-Code der Klasse `Math`

B.2. Terminal-Code der Kopierkartenanwendung

Dieser Abschnitt enthält die Klasse `DepositMachine` des generierten Anwendungscodes für die Ladegeräte. Der gesamte Quellcode der Anwendung ist auf der Projektwebseite[2] zu finden.

B.2.1. Komponentenklasse `DepositMachine` mit den implementierten Protokollschritten der Komponente

```
public class DepositMachine {
private Secret passphrase;
private int money;
private byte stateterminal;
private int amountToLoad;
private Userinterface userinterface;

public void setUserinterface(Userinterface u) {
   this.userinterface = u;}

public DepositMachine() throws TerminalException {
   passphrase = new Secret();
   money = 0;
   stateterminal = StateTerminal.IDLE_TERMINAL;
   amountToLoad = 0;}

public DepositMachine(Secret passphrase, byte stateterminal)
      throws TerminalException {
   this.passphrase = passphrase;
```

[2] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/

```
   this.stateterminal = stateterminal;
   money = 0;
   amountToLoad = 0;}

private void sendMsg(Message msg) {
   userinterface.send((Userinputs) msg);}

private void sendMsg(Message msg, int port)
            throws TerminalException {
   sendMsg(msg, port, false, false);}

private void sendMsg(Message msg, int port,
               Boolean openSessionBeforeSend,
      Boolean closeSessionAfterReceive)
            throws TerminalException {
   Message response = null;
      switch (port) {
      case Ports.DepositMachine2Copycard_default :
         response = sendCardMsg(msg, port);
         if (response != null)
            processMessage(response);
         break;
      default : stop(); }
}

private Message sendCardMsg(Message msg, int port)
            throws TerminalException {
   byte[] encMsg = new
          byte[LengthConstants.MAX_ENCODING_LENGTH_MESSAGES];
   short encMsgLen = Coding.getInstance().encode(
                  (Codeable) msg, encMsg);
   encMsg = ByteArray.subarray(encMsg, 0, encMsgLen);
   byte[] apdu = ByteArray.append(new byte[]{0, 2, 0, 0,
                  0}, encMsg);
   byte[] response;
   try {
      response = transmit(apdu, port);
   } catch (SWTReaderException e) {
      stop();
      return null; //is never executed}

   if (response.length == 2) {
      if (!(response[0] == (byte) 0x90
                  && response[1] == (byte) 0x00))
         throw new TerminalException(
                           "Cardexception: "
      + HandleResult.interpretSW(response));
```

```
      return null;
   } else {
      Coding.getInstance().decodeInit();
      Codeable cResponse = Coding.getInstance()
                                    .decode(response);
      return (Message) cResponse;}
}

public void processRawMessage(byte[] data) {
   try {
      Coding.getInstance().decodeInit();
      Codeable cResponse = Coding.getInstance().decode(data);
      processMessage((Message) cResponse);
   } catch (java.lang.Exception e) {
      System.err.println(
         "Failed decoding data in processRawMessage");}
}

private void processMessage(Message inmsg)
             throws TerminalException {
   switch (inmsg.getCode()) {
      case Code.UINSERTMONEY :
         processUInsertMoney((UInsertMoney) inmsg);
         break;
      case Code.RESAUTHENTICATE :
         processResAuthenticate((ResAuthenticate) inmsg);
         break;
      case Code.RESLOAD :
         processResLoad((ResLoad) inmsg);
         break;
      case Code.UREQUESTBALANCE :
         processURequestBalance((URequestBalance) inmsg);
         break;
      case Code.RESREQUESTBALANCE :
         processResRequestBalance((ResRequestBalance) inmsg);
         break;
      default :
         stop();}
}

public void process(Message msg) throws TerminalException {
   try {
      processMessage(msg);
   } catch (java.lang.Exception e) {
      if (e instanceof TerminalException)
         throw (TerminalException) e;
      else
```

```
        throw new TerminalException(
              "Error when calling process method: " + e);}
}

private final static int MAX_SPLIT_LENGTH = 50;
private final static int MAX_APDU_LENGTH = 254;
private final static int APDU_HEADER_LENGTH = 5;
private final static int APDU_DATA_LENTH = MAX_APDU_LENGTH
      - APDU_HEADER_LENGTH;
private SWTReader[] reader = new SWTReader[2];

public void initReader(SWTReader reader, int port) {
   this.reader[port] = reader;
}

public byte[] encode(Codeable c) throws TerminalException {
   byte[] encMsg = new byte[
                LengthConstants.MAX_ENCODING_LENGTH_MESSAGES];
   short encMsgLen = Coding.getInstance().encode(c, encMsg);
   return ByteArray.subarray(encMsg, 0, encMsgLen);
}

public byte[] transmit(byte[] apdu, int port)
            throws TerminalException,
      SWTReaderException {
   byte[] received = null;
   if (apdu.length <= MAX_APDU_LENGTH) {
      apdu[4] = (byte) (apdu.length - 5);
      received = reader[port].transmit(apdu);
      if (received.length == 2)
         return received;
      received = ByteArray.subarray(received, 0,
                        received.length - 2);
   } else {
      byte[] splittedApdu;
      byte[] apduHeader = ByteArray.subarray(apdu, 0,
                        APDU_HEADER_LENGTH);
      apdu = ByteArray.subarray(apdu, APDU_HEADER_LENGTH);
      int splitCount = apdu.length / APDU_DATA_LENTH;
      if (apdu.length % APDU_DATA_LENTH != 0)
         splitCount++;
      for (int i = 0; i < splitCount; i++) {
         if (i == splitCount - 1) {
            splittedApdu = ByteArray
                  .subarray(apdu, i * APDU_DATA_LENTH);
            apduHeader[2] = (byte) 0;
            apduHeader[4] = (byte) splittedApdu.length;
```

```
            splittedApdu = ByteArray.append(apduHeader,
                                        splittedApdu);
            received = reader[port].transmit(splittedApdu);
            if (received.length == 2)
               return received;
                received = ByteArray.subarray(received, 0,
               received.length - 2);
            } else {
               splittedApdu = ByteArray.subarray(apdu,
                  i * APDU_DATA_LENTH, APDU_DATA_LENTH);
                apduHeader[2] = (byte) 1;
               apduHeader[4] = (byte) splittedApdu.length;
               splittedApdu = ByteArray.append(apduHeader,
                                        splittedApdu);
               received = reader[port].transmit(splittedApdu);
               if (!(received[0] == (byte) 0x90
                    && received[1] == (byte) 0x00)) {
                  throw new TerminalException(
                         "Bad answer from card: part of a "
                       + "long APDU was not answered with "
                       + "OK (0x9000).");
               }
            }
         }

   for (int i = 0; i < MAX_SPLIT_LENGTH; i++) {
      if (received[0] == 0) {
         received = ByteArray.subarray(received, 1);
         return received;
      } else if (received[0] == 1) {
         received = ByteArray.subarray(received, 1);
         byte[] getMore = new byte[]{0, 6, 0, 0, 0, 1, 1};
         byte[] temp_received = reader[port].transmit(getMore);

         temp_received = ByteArray.subarray(temp_received, 0,
               temp_received.length - 2);
         received = ByteArray.append(temp_received[0],
                         received);
         temp_received = ByteArray.subarray(temp_received, 1);
         received = ByteArray.append(received, temp_received);
      } else
         throw new TerminalException(
                 "Bad answer from card: answer to a long APDU "
               + "must begin with 0 or 1.");
   }
   return received;}
```

```
public void initCard_Copycard(int port, int balance,
   Secret passphrase, byte statecard)
       throws TerminalException {
byte cardCode = Code.COPYCARD;
byte[] apdu = {0, 4, 0, 0, 0, cardCode};
apdu = ByteArray.append(apdu,
      ByteArray.append(Code.INT, ByteArray.toByteArray(
                                        balance)));
apdu = ByteArray.append(apdu, (passphrase != null)
      ? encode(passphrase)
      : new byte[]{Code.IGNORE});
apdu = ByteArray.append(apdu, new byte[]{Code.BYTE,
                                        statecard});
try {
   transmit(apdu, port);}
catch (SWTReaderException e) {
   stop();}
}

private void processUInsertMoney(UInsertMoney inmsg)
   throws TerminalException {
   int val = inmsg.getValue();
   if ((val > 0)) {
   setAmountToLoad(val);
   setMoney((money + amountToLoad));
   setStateterminal(StateTerminal.EXPRESAUTH);
   sendMsg(new Authenticate(),
       Ports.DepositMachine2Copycard_default);
   }
   else {
      stop();}
}

private void processResAuthenticate(ResAuthenticate inmsg)
   throws TerminalException {
   Nonce challenge = inmsg.getCardletchallenge();
   if (stateterminal == StateTerminal.EXPRESAUTH) {
      setStateterminal(StateTerminal.EXPRESLOAD);
      AuthData tmp = new AuthData(Constants.LOAD, passphrase,
         challenge, amountToLoad);
      HashedData hashedauth = HashedData.hash(tmp);
      sendMsg(new Load(amountToLoad, hashedauth),
         Ports.DepositMachine2Copycard_default);
   } else {
      stop();}}

private void processResLoad(ResLoad inmsg)
```

```
   throws TerminalException {
   int balance = inmsg.getBalance();
   if (stateterminal == StateTerminal.EXPRESLOAD) {
      setStateterminal(StateTerminal.IDLE_TERMINAL);
      sendMsg(new UResLoad(balance, amountToLoad));
   } else {
      stop();}}

private void processURequestBalance(URequestBalance inmsg)
   throws TerminalException {
   setStateterminal(StateTerminal.IDLE_TERMINAL);
   sendMsg(new RequestBalance(),
       Ports.DepositMachine2Copycard_default);}

private void processResRequestBalance(ResRequestBalance inmsg)
   throws TerminalException {
   int balance = inmsg.getBalance();
   sendMsg(new UShowBalance(balance));}

public Secret getPassphrase() {
   return passphrase;}

public int getMoney() {
   return money;}

public byte getStateterminal() {
   return stateterminal;}

public int getAmountToLoad() {
   return amountToLoad;}

public void setPassphrase(Secret passphrase) {
   this.passphrase = passphrase;}

public void setMoney(int money) {
   this.money = money;}

public void setStateterminal(byte stateterminal) {
   this.stateterminal = stateterminal;}

public void setAmountToLoad(int amountToLoad) {
   this.amountToLoad = amountToLoad;}

protected static void stop() throws TerminalException {
   throw new TerminalException("DepositMachine");}}
```

Listing B.20: Terminal-Code der Klasse `DepositMachine`

C Smart Card-Code für die Sicherheitsdatentypen und Listen

Zusammenfassung: Dieses Kapitel enthält den generierten Smart Card-Code für die in Abschnitt 4.1.2 vorgestellten Sicherheitsdatentypen. Außerdem ist der Smart Card-Code einer Liste beschrieben.

Abschnitt C.1 enthält den Java Card-Code für die Sicherheitsdatentypen, die in Abschnitt 6.2.2 nicht erläutert wurden. In Abschnitt C.2 ist die Java Card-Implementierung einer Liste abgebildet. Der entsprechende Java-Code der Sicherheitsdatentypen sowie einer Liste ist auf der Projektwebseite[1] zu finden.

C.1. Implementierung der Sicherheitsdatentypen

C.1.1. Die Klassen `Key`, `PublicKey` und `PrivateKey`

```
public abstract class Key {
   public boolean isSymmKey() {
      return false;}

   public boolean isPublicKey() {
      return false;}

   public boolean isPrivateKey() {
      return false;}

   public abstract javacard.security.Key toAPIKey();
}
```

Listing C.1: Die abstrakte Klasse `Key`

```
public class PublicKey extends Key implements
                  Codeable, PlainData, SignData {
```

[1] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/

```
   public final static short MODULUSLENGTH = 128;
   public final static short EXPONENTLENGTH = 3;
   private RSAPublicKey apikey;
   public byte[] modulus;
   public byte[] pubexponent = {0x01, 0x00, 0x01};
   private static final byte[] defaultModulus
            = new byte[]{..};

   public PublicKey() {
      modulus = new byte[MODULUSLENGTH];
      apikey = (RSAPublicKey) KeyBuilder.
          buildKey(KeyBuilder.TYPE_RSA_PUBLIC,
            KeyBuilder.LENGTH_RSA_1024,
                                false);
      Arrays.copy(defaultModulus, this.modulus);
      updateKey();}

   public PublicKey(byte[] modulus) {
      this();
      Arrays.copy(modulus, this.modulus);
      updateKey();}

   public boolean equals(PublicKey other) {
      return Arrays.equals(modulus, other.modulus)
         && Arrays.equals(pubexponent, other.pubexponent);}

   public byte getCode() {
      return Code.PUBLICKEY;}

   public void copy(PublicKey p) {
      Arrays.copy(p.modulus, modulus);
      Arrays.copy(p.pubexponent, pubexponent);
      updateKey();}

   public boolean isPublicKey() {
      return true;}

   public void updateKey() {
      apikey.setExponent(pubexponent, (short) 0,
                 EXPONENTLENGTH);
      apikey.setModulus(modulus, (short) 0, MODULUSLENGTH);}

   public javacard.security.Key toAPIKey() {
      return apikey;}
}
```

Listing C.2: Die Klasse `PublicKey`

```
public class PrivateKey extends Key implements
                      Codeable, PlainData, SignData {
   public final static short MODULUSLENGTH = 128;
   public final static short EXPONENTLENGTH = 128;
   private RSAPrivateKey apikey;
   public byte[] modulus;
   public byte[] privexponent;
   private static final byte[] defaultModulus = new byte[]{..};
   private static final byte[] defaultExponent
             = new byte[]{..};

   public PrivateKey() {
      modulus = new byte[MODULUSLENGTH];
      privexponent = new byte[EXPONENTLENGTH];
      apikey = (RSAPrivateKey) KeyBuilder.
          buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
          KeyBuilder.LENGTH_RSA_1024, false);
      Arrays.copy(defaultModulus, this.modulus);
      Arrays.copy(defaultExponent, privexponent);
      updateKey();}

   public PrivateKey(byte[] modulus, byte[] privexp) {
      this();
      Arrays.copy(modulus, this.modulus);
      Arrays.copy(privexp, privexponent);
      updateKey();}

   public boolean equals(PrivateKey other) {
      return Arrays.equals(modulus, other.modulus)
               && Arrays.equals(privexponent,
                           other.privexponent);}

   public byte getCode() {
      return Code.PRIVATEKEY;}

   public void updateKey() {
      apikey.setExponent(privexponent, (short) 0,
              EXPONENTLENGTH);
      apikey.setModulus(modulus, (short) 0, MODULUSLENGTH);}

   public javacard.security.Key toAPIKey() {
      return apikey;}

   public void copy(PrivateKey p) {
      Arrays.copy(p.modulus, modulus);
      Arrays.copy(p.privexponent, privexponent);
      updateKey();}
```

```
   public boolean isPrivateKey() {
      return true;}
}
```

Listing C.3: Die Klasse `PrivateKey`

C.1.2. Die Klasse `Secret`

```
public class Secret implements Codeable, PlainData, SignData {

   public byte[] secret;
   private static final byte[] defaultSecret = new byte[]{..};

   public Secret() {
      secret = new byte[Store.LENGTHOFSECRET];
      Util.arrayCopy(defaultSecret, (short) 0, secret,
            (short) 0,
            (short) (defaultSecret.length
                        > Store.LENGTHOFSECRET
                  ? Store.LENGTHOFSECRET
                  : defaultSecret.length));}

   public Secret(byte[] secret) {
      this.secret = secret;}

   public byte getCode() {
      return Code.SECRET;}

   public boolean equals(Secret other) {
      return Arrays.equals(secret, other.secret);}

   public void copy(Secret from) {
      Arrays.copy(from.secret, this.secret);}
}
```

Listing C.4: Die Klasse `Secret`

C.1.3. Das Interface `HashData` und die Klasse `HashedData`

```
public interface HashData extends Codeable {}
```

Listing C.5: Das Interface `HashData`

```
public class HashedData implements Codeable, HashData {
   public final static short HASHLENGTH = 20;
   public byte[] hashed;
   private static MessageDigest md;
   private static byte[] tohash
             = new byte[Store.COMPUTEDHASHDATALENGTH];
```

```
   public HashedData() {
      hashed = new byte[HASHLENGTH];}

   public void copy(HashedData from) {
      Arrays.copy(from.hashed, this.hashed);}

   public byte getCode() {
      return Code.HASHEDDATA;}

   public boolean equals(HashedData other) {
      return Arrays.equals(hashed, other.hashed);}

   public static HashedData hash(HashData h) {
      if (md == null)
         md = MessageDigest.getInstance(MessageDigest.ALG_SHA,
                     false);
      return Store.newHashedData().insthash(h);}

   public HashedData insthash(HashData h) {
      Coding c = Coding.getInstance();
      c.encode(h, tohash);
      md.doFinal(tohash, (short) 0, c.getEncodingLength(),
                     hashed, (short) 0);
      return this;}
}
```

Listing C.6: Die Klasse `HashedData`

C.1.4. Das Interface `SignData` und die Klasse `SignedData`

```
public interface SignData extends Codeable {}
```

Listing C.7: Das Interface `SignData`

Die Klasse `SignedData` wurde bereits in Abschnitt 6.2.2 erläutert. An dieser Stelle ist deshalb nur die noch fehlende Methode `verify` abgedruckt.

```
public static boolean verify(PublicKey key, SignedData sd,
         SignData s) {
   if (sig == null)
      sig = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1,
                     false);
   return sd.verify(key, s);}

public boolean verify(PublicKey key, SignData s) {
   if (!key.isPublicKey())
      SimpleComm.stop();
   sig.init(key.toAPIKey(), Signature.MODE_VERIFY);
```

```
   Coding.getInstance().encode(s, tmparray);
   short signlength = Coding.getInstance().getEncodingLength();
   return sig.verify(tmparray, (short) 0, signlength, signed,
      (short) 0, signedlength);}
```

Listing C.8: Die Methode `verify` der Klasse `SignedData`

C.1.5. Das Interface `PlainData` und die Klassen `EncData`, `EncDataSymm` und `EncDataAsymm`

```
public interface PlainData extends Codeable{}
```

Listing C.9: Das Interface `PlainData`

```
public abstract class EncData implements Codeable {
   public byte[] encrypted;
   public short plainlength;
   public short enclength;
   public byte plainDataType;
   protected static byte[] tmparray;
   protected Cipher c; }
```

Listing C.10: Die Klasse `EncData`

```
public class EncDataSymm extends EncData {
   public EncDataSymm() {
      encrypted = new byte[Store.MAXENCRYPTLENGTHSYMM];
      tmparray = new byte[Store.MAXENCRYPTLENGTHSYMM];
      c = Cipher.getInstance(Cipher.ALG_DES_CBC_NOPAD, false);}

public static EncDataSymm encrypt(SymmKey key, PlainData d) {
   return Store.newEncDataSymm().instEncrypt(key,d); }

public EncDataSymm instEncrypt(SymmKey key, PlainData d) {
   plainDataType = d.getCode();
   Util.arrayFillNonAtomic(tmparray,(short)0,
         (short)tmparray.length,(byte) 0x0);
   Coding.getInstance().encode(d, tmparray);
   plainlength = Coding.getInstance().getEncodingLength();
   enclength = plainlength % 8 == 0
         ? plainlength
         : (short) (8 + plainlength - plainlength % 8);

   c.init(key.toAPIKey(), Cipher.MODE_ENCRYPT);
   c.doFinal(tmparray,(short)0,enclength,encrypted,(short)0);
   return this; }

private boolean check() {
```

```
   if (!(0 < plainlength && plainlength <= enclength &&
         enclength <= encrypted.length))
      return false;
   return enclength; }

public static PlainData decrypt(SymmKey key, EncDataSymm e) {
   return e.decrypt(key); }

public PlainData decrypt(SymmKey key) {
   if (!check())
      SimpleComm.stop();

   c.init(key.toAPIKey(),Cipher.MODE_DECRYPT);
   c.doFinal(encrypted,(short)0,enclength,tmparray,(short)0);
   return Coding.getInstance().decodePlainData(tmparray,
         plainDataType); }
```

Listing C.11: Smart Card-Code der Klasse `EncDataSymm`

```
public class EncDataAsymm extends EncData {
   private final static short RSABLOCKLENGTH = 128;
   private static RandomData rd;

   public EncDataAsymm() {
      encrypted = new byte[Store.MAXENCRYPTLENGTHASYMM];
      tmparray = new byte[Store.MAXENCRYPTLENGTHASYMM];
      c = Cipher.getInstance(Cipher.ALG_RSA_NOPAD, false);
      if (rd == null)
         rd = RandomData.getInstance(
               RandomData.ALG_SECURE_RANDOM);}

   public byte getCode() {
      return Code.ENCDATAASYMM;}

   public boolean isEncDataAsymm() {
      return true;}

   public static EncDataAsymm encrypt(PublicKey key,
            PlainData d) {
      return Store.newEncDataAsymm().instEncrypt(key, d);}

   public EncDataAsymm instEncrypt(PublicKey key,
            PlainData d) {
      plainDataType = d.getCode();
      Coding.getInstance().encode(d, tmparray);
      plainlength = Coding.getInstance().getEncodingLength();
      enclength = RSABLOCKLENGTH;
```

```
      padBuffer(tmparray, (short) 0, plainlength, enclength);
      c.init(key.toAPIKey(), Cipher.MODE_ENCRYPT);
      c.doFinal(tmparray, (short) 0, plainlength, encrypted,
                  (short) 0);
      return this;}

   private void padBuffer(byte[] buffer, short offset,
         short len, short paddedlen) {
      if (paddedlen != len)
         rd.generateData(buffer, (short) (offset + len),
               (short) (paddedlen - len));}

   private boolean check() {
      if (!(0 < plainlength && plainlength <= enclength
          && enclength <= encrypted.length))
         return false;
      return enclength == RSABLOCKLENGTH;}

   public static PlainData decrypt(PrivateKey key,
            EncDataAsymm e) {
      return e.decrypt(key);}

   public PlainData decrypt(PrivateKey key) {
      if (!check())
         SimpleComm.stop();
      c.init(key.toAPIKey(), Cipher.MODE_DECRYPT);
      c.doFinal(encrypted, (short) 0, enclength, tmparray,
                  (short) 0);
      return Coding.getInstance().decodePlainData(tmparray,
                  plainDataType);}
}
```

Listing C.12: Die Klasse `EncDataAsymm`

C.1.6. Die Klasse `MACData`

```
public class MACData implements Codeable, PlainData, HashData,
         SignData {
   EncDataSymm encData;
   EncDataSymm mac;

   public MACData() {
      encData = new EncDataSymm();
      mac = new EncDataSymm();}

   public static MACData computeMAC(SymmKey encKey,
         SymmKey macKey, PlainData pd) {
```

```
      MACData md = Store.newMACData();
      md.encData = EncDataSymm.encrypt(encKey, pd);
      HashedData hash = HashedData.hash(md.encData);
      md.mac = EncDataSymm.encrypt(macKey, hash);
      return md;}

   public static PlainData decryptMAC(SymmKey encKey,
            SymmKey macKey, MACData md) {
      HashedData hash = HashedData.hash(md.encData);
      EncDataSymm mac = EncDataSymm.encrypt(macKey, hash);
      if (!mac.equals(md.mac))
         SimpleComm.stop();
      return EncDataSymm.decrypt(encKey, md.encData);}

   public byte getCode() {
      return Code.MACDATA;}

   public boolean isMACData() {
      return true;}

   public void copy(MACData o) {
      encData.copy(o.encData);
      mac.copy(o.mac);}

   public boolean equals(MACData o) {
      return o.encData.equals(encData) && o.mac.equals(mac);}
}
```

Listing C.13: Die Klasse `MACData`

C.2. Implementierung von Listen

Nachfolgendes Listing zeigt die Implementierung der Listenklasse `ListOfB`, die in Abschnitt 6.3.1 (Seite 154) als UML-Klasse dargestellt ist.

```
public class ListOfB implements Codeable, PlainData, SignData {
   private short size;
   public B[] elems;
   public short index;

   public ListOfB(short size) {
      this.size = size;
      elems = new B[size];
      for (short i = 0; i < elems.length; i++)
         elems[i] = new B();}

   public short size() {
      return index;}
```

```
public boolean hasFree() {
   return index < elems.length;}

public void add(B e) {
   if (hasFree()) {
      elems[index++].copy(e);}
   else {
      stop();}}

public B at(short index) {
   try { if (index < size()) {
         return elems[index];}
        else{
         stop();
         return null;}
   } catch (ArrayIndexOutOfBoundsException e) {
      stop();
      return null;}
}

public boolean contains(B e) {
   for (short i = 0; i < index; i++) {
      if (elems[i].equals(e))
         return true;
   }
   return false;}

public void remove(B e) {
   removeIndex(containsAndIndex(e));}

private void stop() {
   ISOException.throwIt(
       ISO7816.SW_CONDITIONS_NOT_SATISFIED);}

public void copy(ListOfB from) {
   this.index = from.index;
   for (int i = 0; i < from.index; i++) {
      this.elems[i].copy(from.elems[i]);
   }}

public boolean equals(ListOfB o) {
   if (size != o.size)
      return false;
   for (int i = 0; i < o.size; i++)
      if (!elems[i].equals(o.elems[i]))
         return false;
```

```
    return true;}

  public byte getCode() {
    return Code.LISTOFB;}

  private short containsAndIndex(B e) {
    for (short i = 0; i < index; i++) {
      if (elems[i].equals(e))
        return i;
    }
    return -1;}

  private void removeIndex(short ind) {
    if (ind >= 0 && ind < index) {
      for (short i = ind; (short) (i + 1) < index; i++) {
        elems[i].copy(elems[(short) (i + 1)]);
      }
      index--;}}}
```

Listing C.14: Die Listenklasse `ListOfB`

D Formale Spezifikation der Kopierkartenanwendung

Zusammenfassung: In diesem Kapitel ist die generierte formale Spezifikation der Kopierkartenanwendung angegeben.

ASM =
asm specification ASM
using ASMAUX
procedures
ASM : (agent → int) × (agent → Secret) × (agent → Nonce) × (agent → StateCard) × (agent → int) × (agent → Nonce) × (agent → int) × (agent → Secret) × (agent → int) × (agent → StateTerminal) × (nat → Nonce) × nat × (agent → ports → messagelist) × connections × attackerdataset × bool;
STEP (agent → Secret) × (agent → Secret) × (nat → Nonce) : (agent → int) × (agent → Nonce) × (agent → StateCard) × (agent → int) × (agent → Nonce) × (agent → int) × (agent → int) × (agent → StateTerminal) × nat × (agent → ports → messagelist) × connections × attackerdataset × bool;
DEPOSITMACHINESTEP agent × (agent → Secret) × connections : bool × (agent → int) × (agent → int) × (agent → StateTerminal) × (agent → ports → messagelist) × attackerdataset;
COPYCARDSTEP agent × (agent → Secret) × (nat → Nonce) × connections : bool × (agent → int) × (agent → Nonce) × (agent → StateCard) × nat × (agent → ports → messagelist) × attackerdataset;
COPYINGMACHINESTEP agent × (agent → Secret) × (nat → Nonce) × connections : bool × (agent → int) × (agent → Nonce) × (agent → int) × (agent → StateTerminal) × nat × (agent → ports → messagelist) × attackerdataset;
URequestBalance# agent × connections : (agent → StateTerminal) × (agent → ports → messagelist) × attackerdataset;
RequestBalance# agent × (agent → int) × connections : (agent → StateCard) × (agent → ports → messagelist) × attackerdataset;
ResRequestBalance# message × agent × connections : (agent → ports → messagelist) × attackerdataset;
URequestCopies# message × agent × (nat → Nonce) × connections : bool × (agent → int)

× (agent → Nonce) × (agent → StateTerminal) × nat × (agent → ports → messagelist) × attackerdataset;
Pay# message × agent × (agent → Secret) × connections : bool × (agent → int) × (agent → StateCard) × (agent → ports → messagelist) × attackerdataset;
ResPay# message × agent × (agent → int) × (agent → Nonce) × (agent → Secret) × connections : bool × (agent → int) × (agent → StateTerminal) × (agent → ports → messagelist) × attackerdataset;
UInsertMoney# message × agent × connections : bool × (agent → int) × (agent → int) × (agent → StateTerminal) × (agent → ports → messagelist) × attackerdataset;
Authenticate# agent × (nat → Nonce) × connections : (agent → Nonce) × (agent → StateCard) × nat × (agent → ports → messagelist) × attackerdataset;
ResAuthenticate# message × agent × (agent → int) × (agent → Secret) × connections : bool × (agent → StateTerminal) × (agent → ports → messagelist) × attackerdataset;
Load# message × agent × (agent → Secret) × (agent → Nonce) × connections : bool × (agent → int) × (agent → StateCard) × (agent → ports → messagelist) × attackerdataset;
ResLoad# message × agent × (agent → int) × connections : bool × (agent → StateTerminal) × (agent → ports → messagelist) × attackerdataset;
variables
ad-AuthData-var, tmp-AuthData-var: AuthData;
amountToLoad-int-var, balance-int-var, val-int-var, value-int-var: int;
authhashed-HashedData-var, hashedauth-HashedData-var: HashedData;
challenge-Nonce-var, termchallenge-Nonce-var: Nonce;
stop: bool;
inmsg: message;
state variables Copycard-balance, Copycard-passphrase, Copycard-challenge, Copycard-statecard, CopyingMachine-amountToPay, CopyingMachine-challenge, DepositMachine-amountToLoad, Terminal-passphrase, Terminal-money, Terminal-stateterminal, all-nonces, next-nonce, inputs, connections, attacker-known, stop;
initial state init(Copycard-balance, Copycard-passphrase, Copycard-challenge, Copycard-statecard, CopyingMachine-amountToPay, CopyingMachine-challenge, DepositMachine-amountToLoad, Terminal-passphrase, Terminal-money, Terminal-stateterminal, all-nonces, next-nonce, inputs, connections, attacker-known, stop)
final state stop
asm rule STEP

declaration

ASM(Copycard-balance, Copycard-passphrase, Copycard-challenge, Copycard-statecard, CopyingMachine-amountToPay, CopyingMachine-challenge, DepositMachine-amountToLoad, Terminal-passphrase, Terminal-money, Terminal-stateterminal, all-nonces, next-nonce, inputs, connections, attacker-known, stop)
{
while ¬ stop **do**
STEP
};

STEP(Copycard-passphrase, Terminal-passphrase, all-nonces;**var** Copycard-balance, Copycard-challenge, Copycard-statecard,CopyingMachine-amountToPay, CopyingMachine-challenge, DepositMachine-amountToLoad, Terminal-money, Terminal-stateterminal, next-nonce, inputs, connections, attacker-known, stop)
{
let asm-step = ?, exception_occurred = false **in**
if asm-step = connect **then** CONNECT
else if asm-step = disconnect **then** DISCONNECT
else if asm-step = attacker-agent-step **then** ATTACKER
else if asm-step = user-agent-step **then**
  choose ag **with** is_user(ag) ∧ exagent(ag)
  in USER
  else skip
else if asm-step = Copycard-agent-step **then**
  choose ag **with** is_Copycard(ag) ∧ exagent(ag)
  in COPYCARDSTEP
  else skip
else if asm-step = DepositMachine-agent-step **then**
  choose ag **with** is_DepositMachine(ag) ∧ exagent(ag)
  in DEPOSITMACHINESTEP
  else skip
else if asm-step = CopyingMachine-agent-step **then**
  choose ag **with** is_CopyingMachine(ag) ∧ exagent(ag)
  in COPYINGMACHINESTEP
  else skip;
stop := ?
};

DEPOSITMACHINESTEP(ag, Terminal-passphrase, connections;**var** exception_occurred, DepositMachine-amountToLoad, Terminal-money, Terminal-stateterminal, inputs, attacker-known)
{
choose port **with** is-valid-port(ag, port) ∧ (inputs(ag))(port) ≠ []
in let inmsg = (inputs(ag))(port).first **in**
  {
    inputs := rem(ag, port, inputs);
    if isURequestBalance(inmsg) **then** URequestBalance#
    else if isResRequestBalance(inmsg) **then** ResRequestBalance#
    else if isUInsertMoney(inmsg) **then** UInsertMoney#
    else if isResAuthenticate(inmsg) **then** ResAuthenticate#
    else if isResLoad(inmsg) **then** ResLoad#
    else STOPSTEP(exception_occurred)
  }
else skip
};

```
COPYCARDSTEP(ag, Copycard-passphrase, all-nonces, connections;var exception_occurred,
Copycard-balance, Copycard-challenge, Copycard-statecard, next-nonce, inputs, attacker-known)
{
choose port with is-valid-port(ag, port) ∧ (inputs(ag))(port) ≠ []
in let inmsg = (inputs(ag))(port).first in
  {
    inputs := rem(ag, port, inputs);
    if ¬ is_valid_card_message(inmsg) then skip
    else if isRequestBalance(inmsg) then RequestBalance#
    else if isPay(inmsg) then Pay#
    else if isAuthenticate(inmsg) then Authenticate#
    else if isLoad(inmsg) then Load#
    else STOPSTEP(exception_occurred)
  }
else skip
};

COPYINGMACHINESTEP(ag, Terminal-passphrase, all-nonces, connections;
var exception_occurred, CopyingMachine-amountToPay, CopyingMachine-challenge, Terminal-
money, Terminal-stateterminal, next-nonce, inputs, attacker-known)
{
choose port with is-valid-port(ag, port) ∧ (inputs(ag))(port) ≠ []
in let inmsg = (inputs(ag))(port).first in
  {
    inputs := rem(ag, port, inputs);
    if isURequestBalance(inmsg) then URequestBalance#
    else if isResRequestBalance(inmsg) then ResRequestBalance#
    else if isURequestCopies(inmsg) then URequestCopies#
    else if isResPay(inmsg) then ResPay#
    else STOPSTEP(exception_occurred)
  }
else skip
};

URequestBalance#(ag, connections; var Terminal-stateterminal, inputs, attacker-known)
{
Terminal-stateterminal(ag) := IDLE_TERMINAL;
choose port with    (port = DepositMachine2CopycardDefaultPort ∨ port = CopyingMa-
chine2CopycardDefaultPort)
∧ connected(ag ⊙ port, connections)
in SEND(mkRequestBalance, port, ag, connections; attacker-known, inputs)
else skip
};

RequestBalance#(ag, Copycard-balance, connections;
                    var Copycard-statecard, inputs, attacker-known)
```

{
Copycard-statecard(ag) := IDLE_CARD;
choose port **with** (port = Copycard2DepositMachineDefaultPort ∨
port = Copycard2CopyingMachineDefaultPort)
∧ connected(ag ⊙ port, connections)
in SEND(mkResRequestBalance(Copycard-balance(ag)), port, ag, connections ; attacker-known, inputs)
else skip
};

ResRequestBalance#(inmsg, ag, connections; **var** inputs, attacker-known)
{
let balance-int-var = inmsg.balance **in**
choose port **with** (port = DepositMachine2CardOwnerDefaultPort ∨ port = CopyingMachine2CardOwnerDefaultPort)
∧ connected(ag ⊙ port, connections)
in SEND(mkUShowBalance(balance-int-var),port, ag,connections; attacker-known, inputs)
else skip
};

URequestCopies#(inmsg, ag, all-nonces, connections; **var** exception_occurred, CopyingMachine-amountToPay, CopyingMachine-challenge, Terminal-stateterminal, next-nonce, inputs, attacker-known)
{
let val-int-var = inmsg.value **in**
if val-int-var > 0 **then**
{
CopyingMachine-amountToPay(ag) := val-int-var;
let nonce = [?] **in**
{
NEW-NONCE(all-nonces; next-nonce, nonce);
CopyingMachine-challenge(ag) := nonce
};
Terminal-stateterminal(ag) := EXPRESPAY;
SEND((mkPay(CopyingMachine-amountToPay(ag), CopyingMachine-challenge(ag)),
CopyingMachine2CopycardDefaultPort,ag,connections; attacker-known, inputs)
}
else STOPSTEP(exception_occurred)
};

Pay#(inmsg, ag, Copycard-passphrase, connections; **var** exception_occurred, Copycard-balance, Copycard-statecard, inputs, attacker-known)
{
let value-int-var = inmsg.amount, termchallenge-Nonce-var = inmsg.terminalchallenge **in**
{

```
   Copycard-statecard(ag) := IDLE_CARD;
   if ¬ value-int-var ≤ 0 ∧ shortOverflow(Copycard-balance(ag) - value-int-var) then
      STOPSTEP(exception_occurred)
   else if value-int-var ≤ 0 ∨ Copycard-balance(ag) - value-int-var < 0 then STOPSTEP
   else if shortOverflow(Copycard-balance(ag) - value-int-var) then STOPSTEP
   else {
      Copycard-balance(ag) := Copycard-balance(ag) - value-int-var;
      let tmp-AuthData-var = mkAuthData(PAY, Copycard-passphrase(ag),
      let termchallenge-Nonce-var, value-int-var)
      in
         authhashed-HashedData-var = hash(wrapAuthData2HashData(tmp-AuthData-var))
      in
      SEND(mkResPay(authhashed-HashedData-var),Copycard2CopyingMachineDefaultPort,
      ag, connections; attacker-known, inputs)
   }
}
};

ResPay#(inmsg, ag, CopyingMachine-amountToPay, CopyingMachine-challenge, Terminal-
passphrase, connections;var exception_occurred, Terminal-money, Terminal-stateterminal,
inputs, attacker-known)
{
let authhashed-HashedData-var = inmsg.authcard in
if Terminal-stateterminal(ag) = EXPRESPAY then
   {
      Terminal-stateterminal(ag) := IDLE_TERMINAL;
      if    authhashed-HashedData-var
         = hash(wrapAuthData2HashData(mkAuthData(PAY, Terminal-passphrase(ag),
            CopyingMachine-challenge(ag), CopyingMachine-amountToPay(ag))))
      then
         {
            Terminal-money(ag) := Terminal-money(ag) - CopyingMachine-amountToPay(ag);
            SEND((mkUIssueCopies(CopyingMachine-amountToPay(ag)),
            CopyingMachine2CardOwnerDefaultPort,
            ag, connections; attacker-known, inputs)
         }
      else
      STOPSTEP(exception_occurred)
   }
else STOPSTEP(exception_occurred)
};

UInsertMoney#(inmsg, ag, connections;var exception_occurred, DepositMachine-
amountToLoad, Terminal-money, Terminal-stateterminal, inputs, attacker-known)
{
let val-int-var = inmsg.value in
if val-int-var > 0 then
```

```
  {
    DepositMachine-amountToLoad(ag) := val-int-var;
    Terminal-money(ag) := Terminal-money(ag) + DepositMachine-amountToLoad(ag);
    Terminal-stateterminal(ag) := EXPRESAUTH;
    SEND(mkAuthenticate, DepositMachine2CopycardDefaultPort, ag, connections;
    attacker-known, inputs)
  }
else STOPSTEP(exception_occurred)
};

Authenticate#(ag, all-nonces, connections;var Copycard-challenge, Copycard-statecard, next-
nonce, inputs, attacker-known)
{
let nonce = [?] in
{ NEW-NONCE(all-nonces; next-nonce, nonce);
challenge(ag) := nonce};
Copycard-statecard(ag) := EXPLOAD;
SEND(mkResAuthenticate(Copycard-challenge(ag)), Copycard2DepositMachineDefaultPort,
ag, connections ; attacker-known, inputs)
};

ResAuthenticate#(inmsg, ag, DepositMachine-amountToLoad, Terminal-passphrase, connec-
tions; var exception_occurred, Terminal-stateterminal, inputs, attacker-known)
{
let challenge-Nonce-var = inmsg.cardletchallenge in
if Terminal-stateterminal(ag) = EXPRESAUTH then
  {
    Terminal-stateterminal(ag) := EXPRESLOAD;
    let tmp-AuthData-var = mkAuthData(LOAD, Terminal-passphrase(ag),
        challenge-Nonce-var, DepositMachine-amountToLoad(ag))
    in
    let hashedauth-HashedData-var = hash(wrapAuthData2HashData(tmp-AuthData-var))
    in
    SEND(mkLoad(DepositMachine-amountToLoad(ag), hashedauth-HashedData-var),
    DepositMachine2CopycardDefaultPort, ag, connections ; attacker-known, inputs)
  }
else STOPSTEP(exception_occurred)
};

Load#(inmsg, ag, Copycard-passphrase, Copycard-challenge, connections;
var exception_occurred, Copycard-balance, Copycard-statecard, inputs, attacker-known)
{
let value-int-var = inmsg.amount, hashedauth-HashedData-var = inmsg.authterminal in
if Copycard-statecard(ag) = EXPLOAD then
  {
    Copycard-statecard(ag) := IDLE_CARD;
```

```
      let ad-AuthData-var = mkAuthData(LOAD, Copycard-passphrase(ag),
          Copycard-challenge(ag), value-int-var)
      in
      if   hashedauth-HashedData-var = hash(wrapAuthData2HashData(ad-AuthData-var))
         ∧ value-int-var > 0
         ∧ shortOverflow(value-int-var + Copycard-balance(ag))
      then
         STOPSTEP(exception_occurred)
      else
      if   hashedauth-HashedData-var = hash(wrapAuthData2HashData(ad-AuthData-var))
         ∧ value-int-var > 0
         ∧ value-int-var + Copycard-balance(ag) ≤ 20000
      then
         if shortOverflow(Copycard-balance(ag) + value-int-var) then STOPSTEP
         else {
            Copycard-balance(ag) := Copycard-balance(ag) + value-int-var;
            SEND(mkResLoad(Copycard-balance (ag)), Copycard2DepositMachineDefaultPort,
            ag, connections ; attacker-known, inputs)
         }
      else
      STOPSTEP(exception_occurred)
   }
else STOPSTEP(exception_occurred)
};

ResLoad#(inmsg, ag, DepositMachine-amountToLoad, connections;var exception_occurred,
Terminal-stateterminal, inputs, attacker-known)
{
let balance-int-var = inmsg.balance in
if Terminal-stateterminal(ag) = EXPRESLOAD then
   {
      Terminal-stateterminal(ag) := IDLE_TERMINAL;
      SEND(mkUResLoad(balance-int-var, DepositMachine-amountToLoad(ag)),
      DepositMachine2CardOwnerDefaultPort, ag, connections ; attacker-known, inputs)
   }
else STOPSTEP(exception_occurred)
}
end asm specification
```

ASMAUX =
enrich USER, allInstances, ConnectDisconnect, ManualFuns, accessfuns, constants, Status, generateNonce **with**
predicates init : (agent → int) × (agent → Secret) × (agent → Nonce) × (agent → StateCard) × (agent → int) × (agent → Nonce) × (agent → int) × (agent → Secret) × (agent → int) × (agent → StateTerminal) × (nat → Nonce) × nat × (agent → ports → messagelist) × connections × attackerdataset × bool;

procedures STOPSTEP : bool;
variables
exception_occurred: bool;
Copycard-balance, CopyingMachine-amountToPay, DepositMachine-amountToLoad,
Terminal-money: agent → int;
Copycard-challenge, CopyingMachine-challenge: agent → Nonce;
Copycard-passphrase, Terminal-passphrase: agent → Secret;
Copycard-statecard: agent → StateCard;
Terminal-stateterminal: agent → StateTerminal;
declaration

STOPSTEP(exception_occurred) **{** exception_occurred := true **}**;

end enrich

ManualFuns =
enrich message **with**

end enrich

Status =
data specification
StateCard = EXPLOAD
| IDLE_CARD
;
StateTerminal = IDLE_TERMINAL
| EXPRESPAY
| EXPRESAUTH
| EXPRESLOAD
;
variables
a_StateCard: StateCard;
a_StateTerminal: StateTerminal;
end data specification

USER =
enrich SEND **with**
procedures
USER-READ agent × connections : (agent → ports → messagelist);
USER-SEND agent × connections : attackerdataset × (agent → ports → messagelist);
USER agent × connections : attackerdataset × (agent → ports → messagelist);
variables connections: connections;
declaration

USER-READ(ag, connections; **var** inputs)
{

choose port **with** connected(ag ⊙ port, connections)
in (inputs(ag))(port) := []
else skip
};

USER-SEND(ag, connections; **var** attacker-known, inputs)
{
choose port, msg **with** connected(ag ⊙ port, connections) ∧ is_user_message(msg)
in SEND(msg, port, ag, connections**;** attacker-known, inputs)
else skip
};

USER(ag, connections; **var** attacker-known, inputs)
{
choose user-step = ? **in**
if user-step = send **then** USER-SEND(ag, connections**;** attacker-known, inputs)
else if user-step = read **then** USER-READ(ag, connections**;** inputs)
}

end enrich

---

accessfuns =
enrich messageaccessfuns **with**
functions

. .instruction:= .	:	AuthData × int	→	AuthData;
. .passphrase:= .	:	AuthData × Secret	→	AuthData;
. .challenge:= .	:	AuthData × Nonce	→	AuthData;
. .amount:= .	:	AuthData × int	→	AuthData;

axioms
a_AuthData .instruction:= i
= mkAuthData(i, a_AuthData.passphrase, a_AuthData.challenge, a_AuthData.amount);

a_AuthData .passphrase:= a_Secret
= mkAuthData(a_AuthData.instruction, a_Secret, a_AuthData.challenge,
a_AuthData.amount);

a_AuthData .challenge:= a_Nonce
= mkAuthData(a_AuthData.instruction, a_AuthData.passphrase, a_Nonce,
a_AuthData.amount);

a_AuthData .amount:= i
= mkAuthData(a_AuthData.instruction, a_AuthData.passphrase, a_AuthData.challenge,
i);

end enrich

---

allInstances =
enrich agents, natlist **with**
constants
CopycardAllInstances : agents;

DepositMachineAllInstances : agents;
CopyingMachineAllInstances : agents;

functions

count-m-to-n	:	nat × nat	→	natlist ;
mkCopycardAllInstances	:	natlist	→	agents ;
mkDepositMachineAllInstances	:	natlist	→	agents ;
mkCopyingMachineAllInstances	:	natlist	→	agents ;

predicates

exCopycardAgents	:	agents;
exDepositMachineAgents	:	agents;
exCopyingMachineAgents	:	agents;

axioms

count-m-to-n-done : n ≤ m → count-m-to-n(m, n) = [];

count-m-to-n-rec : m < n → count-m-to-n(m, n) = m + count-m-to-n(m + 1, n);

mkCopycardAllInstances-empty : mkCopycardAllInstances([]) = [];

mkCopycardAllInstances-rec :
mkCopycardAllInstances(m ' + nats) = Copycard(m) + mkCopycardAllInstances(nats);

CopycardAllInstances-def :
CopycardAllInstances = mkCopycardAllInstances(count-m-to-n(0, NUMOFCOPYCARDS));

exCopycardAgents-def :
exCopycardAgents(agents) ↔ (∀ ag. ag ∈ agents → exagent(ag) ∧ is_Copycard(ag));

mkDepositMachineAllInstances-empty : mkDepositMachineAllInstances([]) = [];

mkDepositMachineAllInstances-rec :
mkDepositMachineAllInstances(m ' + nats)
= DepositMachine(m) + mkDepositMachineAllInstances(nats);

DepositMachineAllInstances-def :
DepositMachineAllInstances
= mkDepositMachineAllInstances(count-m-to-n(0, NUMOFDEPOSITMACHINES));

exDepositMachineAgents-def :
exDepositMachineAgents(agents) ↔ (∀ ag. ag ∈ agents →
exagent(ag) ∧ is_DepositMachine(ag));

mkCopyingMachineAllInstances-empty : mkCopyingMachineAllInstances([]) = [];

mkCopyingMachineAllInstances-rec :
mkCopyingMachineAllInstances(m ' + nats)
= CopyingMachine(m) + mkCopyingMachineAllInstances(nats);

CopyingMachineAllInstances-def :
CopyingMachineAllInstances
= mkCopyingMachineAllInstances(count-m-to-n(0, NUMOFCOPYINGMACHINES));

exCopyingMachineAgents-def :
exCopyingMachineAgents(agents) ↔ (∀ ag. ag ∈ agents →
exagent(ag) ∧ is_CopyingMachine(ag));

count-m-to-n-dups : $\neg$ dups(count-m-to-n(m, n));

count-m-to-n-in : m $\in$ count-m-to-n(n, n_0) $\leftrightarrow \neg \neg$ (n $\leq$ m $\wedge$ m $<$ n_0);

CopycardAllInstances-dups : $\neg$ dups(CopycardAllInstances);

CopycardAllInstances-in :
ag $\in$ CopycardAllInstances $\leftrightarrow \neg \neg$ (exagent(ag) $\wedge$ is_Copycard(ag));

exCopycardAgents-CopycardAllInstances : exCopycardAgents(CopycardAllInstances);

exCopycardAgents-not : $\neg$ exagent(ag) $\wedge$ ag $\in$ agents $\rightarrow \neg$ exCopycardAgents(agents);

exCopycardAgents-not : $\neg$ is_Copycard(ag) $\wedge$ ag $\in$ agents $\rightarrow \neg$ exCopycardAgents(agents);

exCopycardAgents-rec :
exCopycardAgents(ag ' + agents)
$\leftrightarrow \neg \neg$ (exagent(ag) $\wedge$ is_Copycard(ag) $\wedge$ exCopycardAgents(agents));

mkCopycardAllInstances-dups : dups(mkCopycardAllInstances(nats)) $\leftrightarrow$ dups(nats);

mkCopycardAllInstances-in :
ag $\in$ mkCopycardAllInstances(nats) $\leftrightarrow \neg \neg$ (is_Copycard(ag) $\wedge$ ag.name $\in$ nats);

DepositMachineAllInstances-dups : $\neg$ dups(DepositMachineAllInstances);

DepositMachineAllInstances-in :
ag $\in$ DepositMachineAllInstances $\leftrightarrow \neg \neg$ (exagent(ag) $\wedge$ is_DepositMachine(ag));

exDepositMachineAgents-DepositMachineAllInstances :
exDepositMachineAgents(DepositMachineAllInstances);

exDepositMachineAgents-not :
$\neg$ exagent(ag) $\wedge$ ag $\in$ agents $\rightarrow \neg$ exDepositMachineAgents(agents);

exDepositMachineAgents-not :
$\neg$ is_DepositMachine(ag) $\wedge$ ag $\in$ agents $\rightarrow \neg$ exDepositMachineAgents(agents);

exDepositMachineAgents-rec :
exDepositMachineAgents(ag ' + agents)
$\leftrightarrow \neg \neg$ (exagent(ag) $\wedge$ is_DepositMachine(ag) $\wedge$ exDepositMachineAgents(agents));

mkDepositMachineAllInstances-dups :
dups(mkDepositMachineAllInstances(nats)) $\leftrightarrow$ dups(nats);

mkDepositMachineAllInstances-in :
ag $\in$ mkDepositMachineAllInstances(nats) $\leftrightarrow \neg \neg$ (is_DepositMachine(ag) $\wedge$
ag.name $\in$ nats);

CopyingMachineAllInstances-dups : $\neg$ dups(CopyingMachineAllInstances);

CopyingMachineAllInstances-in :
ag $\in$ CopyingMachineAllInstances $\leftrightarrow \neg \neg$ (exagent(ag) $\wedge$ is_CopyingMachine(ag));

exCopyingMachineAgents-CopyingMachineAllInstances :
exCopyingMachineAgents(CopyingMachineAllInstances);

exCopyingMachineAgents-not :
$\neg$ exagent(ag) $\wedge$ ag $\in$ agents $\rightarrow \neg$ exCopyingMachineAgents(agents);

exCopyingMachineAgents-not :
$\neg$ is_CopyingMachine(ag) $\wedge$ ag $\in$ agents $\rightarrow \neg$ exCopyingMachineAgents(agents);

exCopyingMachineAgents-rec :
exCopyingMachineAgents(ag ' + agents)
$\leftrightarrow \neg \neg$ (exagent(ag) $\wedge$ is_CopyingMachine(ag) $\wedge$ exCopyingMachineAgents(agents));

mkCopyingMachineAllInstances-dups :
dups(mkCopyingMachineAllInstances(nats)) $\leftrightarrow$ dups(nats);

mkCopyingMachineAllInstances-in :
ag $\in$ mkCopyingMachineAllInstances(nats) $\leftrightarrow \neg \neg$ (is_CopyingMachine(ag) $\wedge$
ag.name $\in$ nats);

end enrich

---

constants =
enrich int-pair **with**
constants
PAY : int;
LOAD : int;

axioms

PAY : PAY = 1;

LOAD : LOAD = 2;

diff : PAY $\neq$ LOAD;

end enrich

---

generateNonce =
enrich Nonce, int-pair **with**
predicates
. $\in$. : Nonce $\times$ (nat $\rightarrow$ Nonce);
dups : (nat $\rightarrow$ Nonce);
in-rest : Nonce $\times$ (nat $\rightarrow$ Nonce) $\times$ nat;
old : Nonce $\times$ (nat $\rightarrow$ Nonce) $\times$ nat;
procedures generateNonce#(nat $\rightarrow$ Nonce) : nat $\times$ Nonce;
variables
all-nonces, all-nonces$_0$: nat $\rightarrow$ Nonce;
next-nonce: nat;
declaration

generateNonce#(all-nonces; **var** next-nonce, a_Nonce)
{
a_Nonce := all-nonces(next-nonce); next-nonce := next-nonce + 1
}

axioms

nonce-in-rest-def :
in-rest(a_Nonce, all-nonces, next-nonce) ↔ (∃ n. n ≥ next-nonce ∧ all-nonces(n) = a_Nonce);

nonce-old-def :
old(a_Nonce, all-nonces, next-nonce) ↔ ¬ ∃ n. n ≥ next-nonce ∧ all-nonces(n) = a_Nonce;

nonce-in-def : a_Nonce ∈ all-nonces ↔ (∃ n. a_Nonce = all-nonces(n));

nonce-dups-def : dups(all-nonces) ↔ (∃ n, n_0. n ≠ n_0 ∧ all-nonces(n) = all-nonces(n_0));

old-next-nonce :
old(a_Nonce, all-nonces, next-nonce) → old(a_Nonce, all-nonces, next-nonce + 1);

old-next-nonce-nodups :
¬ dups(all-nonces) → old(all-nonces(next-nonce), all-nonces, next-nonce + 1);

old-not-this-nonce : ¬ old(all-nonces(next-nonce), all-nonces, next-nonce);

end enrich

SEND =
enrich ConnectDisconnect **with**
procedures SEND : message× ports× agent× connections : attackerdataset × (agent → ports → messagelist);
variables
inputs: agent → ports → messagelist;
input: ports → messagelist;
attacker-known: attackerdataset;
outport, rem-port: ports;
outmsg: message;
rem-agent: agent;
declaration

SEND(outmsg, outport, ag, connections; **var** attacker-known, inputs)
{
var conn **with** is-endpoint(ag ⊙ outport, conn) ∧ conn ∈ connections
in var rem-agent = other-endpoint(ag ⊙ outport, conn).agent,
rem-port = other-endpoint(ag ⊙ outport, conn).port
in
{
if attacker-can-read(conn) **then** attacker-known := attacker-known ∫+ outmsg;
if ¬ is_attacker(rem-agent) **then**
inputs := add(rem-agent, rem-port, outmsg, inputs)
}
else skip
}

end enrich

---

agents =
actualize list-perm **with** exagent **by morphism**

elem → agent; list → agents; a → ag; a_0 → ag_0; b → ag_1; c → ag_2; x → agents; x_0 → $agents_0$; y → $agents_1$; z → $agents_2$; y_0 → $agents_3$; z_0 → $agents_4$; x_1 → $agents_5$; y_1 → $agents_6$; z_1 → $agents_7$; x_2 → $agents_8$; y_2 → $agents_9$; z_2 → $agents_{10}$

end actualize

---

messageaccessfuns =

enrich Attacker **with**

functions

. .amountToLoad:= .	:	message × int	→	message	**prio** 9;
. .balance:= .	:	message × int	→	message	**prio** 9;
. .value:= .	:	message × int	→	message	**prio** 9;
. .cardletchallenge:= .	:	message × Nonce	→	message	**prio** 9;
. .authcard:= .	:	message × HashedData	→	message	**prio** 9;
. .terminalchallenge:= .	:	message × Nonce	→	message	**prio** 9;
. .amount:= .	:	message × int	→	message	**prio** 9;
. .authterminal:= .	:	message × HashedData	→	message	**prio** 9;

axioms

isLoad(msg) → msg .authterminal:= a_HashedData = mkLoad(msg.amount, a_HashedData);

isLoad(msg) → msg .amount:= i = mkLoad(i, msg.authterminal);

isPay(msg) → msg .amount:= i = mkPay(i, msg.terminalchallenge);

isPay(msg) → msg .terminalchallenge:= a_Nonce = mkPay(msg.amount, a_Nonce);

isResRequestBalance(msg) → msg .balance:= i = mkResRequestBalance(i);

isResPay(msg) → msg .authcard:= a_HashedData = mkResPay(a_HashedData);

isResAuthenticate(msg) → msg .cardletchallenge:= a_Nonce = mkResAuthenticate(a_Nonce);

isURequestCopies(msg) → msg .value:= i = mkURequestCopies(i);

isUInsertMoney(msg) → msg .value:= i = mkUInsertMoney(i);

isUIssueCopies(msg) → msg .value:= i = mkUIssueCopies(i);

isUShowBalance(msg) → msg .value:= i = mkUShowBalance(i);

isResLoad(msg) → msg .balance:= i = mkResLoad(i);

isUResLoad(msg) → msg .balance:= i = mkUResLoad(i, msg.amountToLoad);

isUResLoad(msg) → msg .amountToLoad:= i = mkUResLoad(msg.balance, i);

end enrich

ConnectDisconnect =

enrich Attacker **with**

predicates

disconnect-possible	:	connection × connections × (agent → ports → messagelist);
connect-possible	:	connection × connections × (agent → ports → messagelist);

procedures
CONNECT : connections × (agent → ports → messagelist);
DISCONNECT : connections × (agent → ports → messagelist);
variables
input: ports → messagelist;
inputs: agent → ports → messagelist;
declaration

CONNECT(connections, inputs)
{
var conn **with** connect-possible(conn, connections, inputs)
in connections := connections ++ conn
else skip
};

DISCONNECT(connections, inputs)
{
var conn **with** disconnect-possible(conn, connections, inputs)
in {
connections := connections – conn;
(inputs(conn.endpoint1.agent))(conn.endpoint1.port) := [];
(inputs(conn.endpoint2.agent))(conn.endpoint2.port) := []
}
else skip
}
axioms

connect-possible-def :
connect-possible(conn, connections, inputs)
↔ connect-possible(conn, connections)
∧ (inputs(conn.endpoint1.agent))(conn.endpoint1.port) = []
∧ (inputs(conn.endpoint2.agent))(conn.endpoint2.port) = [];

disconnect-possible-def :
disconnect-possible(conn, connections, inputs)
↔ conn ∈ connections
∧ (inputs(conn.endpoint1.agent))(conn.endpoint1.port) = []
∧ (inputs(conn.endpoint2.agent))(conn.endpoint2.port) = [];

valid-conns-add :
valid-conns(connections) ∧ connect-possible(conn, connections, inputs)
→ valid-conns(connections ++ conn);

end enrich

Attacker =
enrich Generate, AttackerAbilities, inputs, asmstep **with**

procedures
ATTACKER-SEND connections × attackerdataset : (agent → ports → messagelist);
ATTACKER-SUPPRESS connections : (agent → ports → messagelist);
ATTACKER-DISCONNECT : connections × (agent → ports → messagelist);
ATTACKER connections × attackerdataset : (agent → ports → messagelist);
variables
inputs: agent → ports → messagelist;
input: ports → messagelist;
attacker-known: attackerdataset;
declaration
ATTACKER-SEND(connections, attacker-known; **var** inputs)
{
var msg, endp **with** attacker-known ≫ amessage(msg) ∧ attacker-can-send(endp, connections)
in (inputs(endp.agent))(endp.port) := (inputs(endp.agent))(endp.port) + msg '
else skip
};
ATTACKER-SUPPRESS(connections; **var** inputs)
{
var endp **with** attacker-can-suppress(endp, connections) ∧ (inputs(endp.agent))(endp.port) ≠ []
in inputs := rem(endp.agent, endp.port, inputs)
else skip
};
ATTACKER-DISCONNECT(connections, inputs)
{
var conn **with** conn ∈ connections ∧ attacker-can-suppress(conn, connections)
in {
connections := connections – conn;
(inputs(conn.endpoint1.agent))(conn.endpoint1.port) := [];
(inputs(conn.endpoint2.agent))(conn.endpoint2.port) := []
}
else skip
};
ATTACKER(connections, attacker-known; **var** inputs)
{
var attacker-step = ? **in**
if attacker-step = attacker-send **then**
ATTACKER-SEND(connections, attacker-known**;** inputs)
else if attacker-step = attacker-suppress **then** ATTACKER-SUPPRESS(connections**;** inputs)
else ATTACKER-DISCONNECT(**;** connections, inputs)
}
end enrich

AttackerAbilities =
enrich connectionPreds **with**
predicates

eavesdrop	:	connection;
suppress	:	connection;
send	:	connection;
attacker-can-read	:	connection;
attacker-can-send	:	connections;
attacker-can-send	:	endpoint × connections;
attacker-can-suppress	:	endpoint × connections;
attacker-can-suppress	:	connection × connections;
attacker-can-suppress	:	connections;

axioms

attacker-can-read :
attacker-can-read(conn) ↔ is-endpoint(attacker, conn) ∨ eavesdrop(conn);

attacker-can-send-conn :
attacker-can-send(connections) ↔ (∃ endp. attacker-can-send(endp, connections));

attacker-can-send :
attacker-can-send(endp, connections)
↔ ¬ is_attacker(endp.agent)
∧ (∃ conn. conn ∈ connections ∧ is-endpoint(endp, conn) ∧ send(conn));

attacker-can-suppress-endpoint :
attacker-can-suppress(endp, connections)
↔ ∃ conn. conn ∈ connections ∧ is-endpoint(endp, conn) ∧ suppress(conn);

attacker-can-suppress-conn :
attacker-can-suppress(conn, connections) ↔ conn ∈ connections ∧ suppress(conn);

attacker-can-suppress-any :
attacker-can-suppress(connections) ↔ (∃ endp. attacker-can-suppress(endp, connections));

eavesdrop :
eavesdrop(conn)
↔ conn.endpoint1.port = CopyingMachine2CopycardDefaultPort
∧ conn.endpoint2.port = Copycard2CopyingMachineDefaultPort
∨ conn.endpoint1.port = DepositMachine2CopycardDefaultPort
∧ conn.endpoint2.port = Copycard2DepositMachineDefaultPort;

suppress :
suppress(conn)
↔ conn.endpoint1.port = CopyingMachine2CopycardDefaultPort
∧ conn.endpoint2.port = Copycard2CopyingMachineDefaultPort
∨ conn.endpoint1.port = DepositMachine2CopycardDefaultPort
∧ conn.endpoint2.port = Copycard2DepositMachineDefaultPort;

send :
send(conn)
↔ conn.endpoint1.port = CopyingMachine2CopycardDefaultPort
∧ conn.endpoint2.port = Copycard2CopyingMachineDefaultPort

∨ conn.endpoint1.port = DepositMachine2CopycardDefaultPort
∧ conn.endpoint2.port = Copycard2DepositMachineDefaultPort;

end enrich

Generate =
enrich AddAttackerdata **with**
predicates . ≫ . : attackerdataset × attackerdata;

axioms

nonce : adset ≫ anonce(a_Nonce) ↔ anonce(a_Nonce) ∈ adset;

secret : adset ≫ asecret(a_Secret) ↔ asecret(a_Secret) ∈ adset;

hasheddata :
adset ≫ ahasheddata(a_HashedData)
↔ ahasheddata(a_HashedData) ∈ adset ∨ adset ≫ ahashdata(a_HashedData.hash);

generate-Load :
adset ≫ amessage(mkLoad(i, a_HashedData)) ↔ adset ≫ ahasheddata(a_HashedData);

generate-RequestBalance : adset ≫ amessage(mkRequestBalance);

generate-InvalidMessage : adset ≫ amessage(mkInvalidMessage);

generate-Pay : adset ≫ amessage(mkPay(i, a_Nonce)) ↔ adset ≫ anonce(a_Nonce);

generate-ResRequestBalance : adset ≫ amessage(mkResRequestBalance(i));

generate-ResPay :
adset ≫ amessage(mkResPay(a_HashedData)) ↔ adset ≫ ahasheddata(a_HashedData);

generate-ResAuthenticate :
adset ≫ amessage(mkResAuthenticate(a_Nonce)) ↔ adset ≫ anonce(a_Nonce);

generate-URequestBalance : adset ≫ amessage(mkURequestBalance);

generate-URequestCopies : adset ≫ amessage(mkURequestCopies(i));

generate-UInsertMoney : adset ≫ amessage(mkUInsertMoney(i));

generate-Authenticate : adset ≫ amessage(mkAuthenticate);

generate-UIssueCopies : adset ≫ amessage(mkUIssueCopies(i));

generate-UShowBalance : adset ≫ amessage(mkUShowBalance(i));

generate-ResLoad : adset ≫ amessage(mkResLoad(i));

generate-UResLoad : adset ≫ amessage(mkUResLoad(i, i_0));

authData4data :
adset ≫ adata(wrapAuthData2data(mkAuthData(i, a_Secret, a_Nonce, i_0)))
↔ adset ≫ asecret(a_Secret) ∧ adset ≫ anonce(a_Nonce);

authData4HashData :
adset ≫ ahashdata(wrapAuthData2HashData(mkAuthData(i, a_Secret, a_Nonce, i_0)))
↔ adset ≫ asecret(a_Secret) ∧ adset ≫ anonce(a_Nonce);

user-messages-known : is_user_message(msg) → adset ≫ amessage(msg);

generate-max-in : ad ∈ adset ∧ max∫ adset → adset ≫ ad;

generate-max-in : ad ∈ adset ∧ max∫ adset → adset$_0$ ∪ adset ≫ ad;

generate-add-message : ¬ adset ++ ad ≫ amessage(msg) → ¬ adset ≫ amessage(msg);

generate-add : ¬ adset ++ ad ≫ ad$_0$ → ¬ adset ≫ ad$_0$;

generate-subset : adset ≫ ad ∧ adset ⊆ adset$_0$ → adset$_0$ ≫ ad;

generate-message : sub∫(amessage(msg), adset) ⊆ ∅ ++ ad → adset ++ ad ≫ amessage(msg);

generate-message :
sub∫(amessage(msg), adset) ⊆ ∅ ++ ad ++ ad$_0$ → adset ++ ad ++ ad$_0$ ≫ amessage(msg);

generate-more : ¬ adset ++ ad ++ ad$_0$ ≫ ad$_1$ → ¬ adset ≫ ad$_1$;

generate-more : ¬ adset ++ ad ≫ ad$_1$ → ¬ adset ≫ ad$_1$;

generate-message-subset :
sub∫(amessage(msg), adset$_1$) ⊆ adset ∧ adset$_1$ ∪ adset ⊆ adset$_0$ → adset$_0$ ≫ amessage(msg);

generate-hasheddata :
ahasheddata(a_HashedData) ∈ adset → adset ≫ ahasheddata(a_HashedData);

generate-hasheddata :
ahasheddata(a_HashedData) ∈ adset → adset$_0$ ∪ adset ≫ ahasheddata(a_HashedData);

generate-nonce : anonce(a_Nonce) ∈ adset → adset ≫ anonce(a_Nonce);

generate-nonce : anonce(a_Nonce) ∈ adset → adset$_0$ ∪ adset ≫ anonce(a_Nonce);

generate-secret : asecret(a_Secret) ∈ adset → adset ≫ asecret(a_Secret);

generate-secret : asecret(a_Secret) ∈ adset → adset$_0$ ∪ adset ≫ asecret(a_Secret);

end enrich

---

asmstep =
data specification
asm-step = Copycard-agent-step
| DepositMachine-agent-step
| CopyingMachine-agent-step
| connect
| disconnect
| user-agent-step
| attacker-agent-step
;
attacker-step = attacker-send
| attacker-suppress
| attacker-disconnect
;
user-step = read
| send

;
variables
asm-step, asm-step$_0$: asm-step;
attacker-step, attacker-step$_0$: attacker-step;
user-step, user-step$_0$: user-step;
end data specification

---

inputs =
enrich messagelist, connectionPreds **with**
functions
add : agent × ports × message × (agent → ports → messagelist) → agent → ports → messagelist;
rem : agent × ports × (agent → ports → messagelist) → agent → ports → messagelist;
predicates
input-available : agent × (agent → ports → messagelist);
msgininputs : message × (agent → ports → messagelist);
variables
inputs, inputs$_0$, inputs$_1$, inputs$_2$: agent → ports → messagelist;
portmsgs, portmsgs$_0$, portmsgs$_1$, portmsgs$_2$: ports → messagelist;
axioms
add-def :
add(ag, port, msg, inputs) = [(inputs, ag, [(inputs(ag), port, (inputs(ag))(port) + msg));
rem-def :
rem(ag, port, inputs) = [(inputs, ag, [(inputs(ag), port, (inputs(ag))(port).rest));
inputavailable : input-available(ag, inputs) ↔ (∃ port. (inputs(ag))(port) ≠ []);
msgininputs : msgininputs(msg, inputs) ↔ (∃ ag, port. msg ∈ (inputs(ag))(port));
inputs-identity : [(inputs, ag, inputs(ag)) = inputs;
inputs-identity : [(portmsgs, port, portmsgs(port)) = portmsgs;
inputs-identity : portmsgs(port) = msgs → [(portmsgs, port, msgs) = portmsgs;
inputs-identity : inputs(ag) = portmsgs → [(inputs, ag, portmsgs) = inputs;
inputs-identity :
ag ≠ ag$_0$
→ [([(inputs, ag, [(portmsgs, port, msgs)), ag$_0$, inputs(ag$_0$))
= [(inputs, ag, [(portmsgs, port, msgs));
inputs-override : [([(inputs, ag, portmsgs), ag, portmsgs$_0$) = [(inputs, ag, portmsgs$_0$);
inputs-override : [([(portmsgs, port, msgs), port, msgs$_0$) = [(portmsgs, port, msgs$_0$);
inputs-switch :
ag ≠ ag$_0$ → [([(inputs, ag, portmsgs), ag$_0$, portmsgs$_0$) = [([(inputs, ag$_0$, portmsgs$_0$), ag, portmsgs);

inputs-switch :
port $\neq$ port$_0$ $\rightarrow$ [([(portmsgs, port, msgs), port$_0$, msgs$_0$) = [([(portmsgs, port$_0$, msgs$_0$), port, msgs);

msgininputs-add : msgininputs(msg, add(ag, port, msg, inputs));

msgininputs-notin : $\neg$ msgininputs(msg, inputs) $\rightarrow$ $\neg$ msg $\in$ (inputs(ag))(port);

msgininputs-notin :
$\neg$ msgininputs(msg, inputs) $\wedge$ msg $\in$ msgs$_0$ $\rightarrow$ (inputs(ag))(port) $\neq$ msg$_0$ ' + msgs$_0$;

end enrich

AddAttackerdata =
enrich Analyze, messagelist **with**
functions

. $\int$+ .	:	attackerdataset $\times$ message	$\rightarrow$	attackerdataset	**prio** 9;
. $\int$+ .	:	attackerdataset $\times$ messagelist	$\rightarrow$	attackerdataset	**prio** 9;

axioms

add-one : adset $\int$+ msg = adset $\cup$ sub$\int$(amessage(msg), adset);

add-empty : adset $\int$+ [] = $\int$ adset;

adds-rec : adset $\int$+ msg ' + msgs = (adset $\cup$ sub$\int$(amessage(msg), adset)) $\int$+ msgs;

end enrich

---

connectionPreds =
enrich exagent **with**
predicates

conn-ok	:	connection;
valid-conn	:	connection;
valid-conns	:	connections;
connected	:	endpoint $\times$ connections;
connected	:	agent $\times$ connections;
connect-possible	:	connection $\times$ connections;

variables conns, connections: connections;

axioms

conn-ok :
conn-ok(conn)
$\leftrightarrow$ endpoint-ok(conn.endpoint1)
$\wedge$ endpoint-ok(conn.endpoint2)
$\wedge$ (conn.endpoint1.port = CardOwner2CopyingMachineDefaultPort
$\wedge$ conn.endpoint2.port = CopyingMachine2CardOwnerDefaultPort
$\vee$ conn.endpoint1.port = CardOwner2DepositMachineDefaultPort
$\wedge$ conn.endpoint2.port = DepositMachine2CardOwnerDefaultPort
$\vee$ conn.endpoint1.port = CopyingMachine2CopycardDefaultPort

$\wedge$ conn.endpoint2.port = Copycard2CopyingMachineDefaultPort
$\vee$ conn.endpoint1.port = DepositMachine2CopycardDefaultPort
$\wedge$ conn.endpoint2.port = Copycard2DepositMachineDefaultPort);

valid-conn :
valid-conn(conn)
$\leftrightarrow$ conn-ok(conn) $\wedge$ exagent(conn.endpoint1.agent) $\wedge$ exagent(conn.endpoint2.agent);

valid-conns : valid-conns(connections) $\leftrightarrow$ ($\forall$ conn. conn $\in$ connections $\rightarrow$ valid-conn(conn));

connected-endpoint-def :
connected($endp_1$, connections)
$\leftrightarrow \exists$ $endp_2$. mk-connection($endp_1$, $endp_2$) $\in$ connections
$\vee$ mk-connection($endp_2$, $endp_1$) $\in$ connections;

connected-agent-def :
connected(ag, connections) $\leftrightarrow$ ($\exists$ port. connected(ag $\odot$ port, connections));

connect-possible :
connect-possible(conn, connections)
$\leftrightarrow$ valid-conn(conn)
$\wedge \neg$ connected(conn.endpoint1, connections)
$\wedge \neg$ connected(conn.endpoint2, connections)
$\wedge$ (is_statefulService(conn.endpoint2.agent)
$\rightarrow \neg$ connected(conn.endpoint2.agent, connections));

valid-conns-add :
valid-conns(connections) $\wedge$ connect-possible(conn, connections)
$\rightarrow$ valid-conns(connections ++ conn);

valid-conns-delete : valid-conns(connections) $\rightarrow$ valid-conns(connections – conn);

end enrich

Analyze =
enrich submessages **with**
functions $\int$. : attackerdataset $\rightarrow$ attackerdataset ;
predicates max$\int$. : attackerdataset;

axioms
add-max : max$\int$ adset $\rightarrow$ $\int$ adset = adset;

add-rec : ad $\in$ adset $\rightarrow$ $\int$ adset = $\int$(adset $\cup$ sub$\int$(ad, adset));

maxknown : max$\int$ adset $\leftrightarrow$ ($\forall$ ad. ad $\in$ adset $\rightarrow$ sub$\int$(ad, adset) $\subseteq$ adset);

add-subset : adset $\subseteq$ $\int$ adset;

integ-is-max : max$\int$ $\int$ adset;

max-known-secret-01 : max$\int$ adset $\rightarrow$ max$\int$(adset ++ asecret(a_Secret));

max-known-secret :
max$\int$ adset $\rightarrow$ $\int$(adset ++ asecret(a_Secret)) = adset ++ asecret(a_Secret);

max-known-nonce : max$\int$ adset $\rightarrow$ $\int$(adset ++ anonce(a_Nonce)) =
adset ++ anonce(a_Nonce);

max-known-nonce-01 : max$\int$ adset $\rightarrow$ max$\int$(adset ++ anonce(a_Nonce));

max-known-hasheddata :
max$\int$ adset $\rightarrow$
$\int$(adset ++ ahasheddata(a_HashedData)) = adset ++ ahasheddata(a_HashedData);

max-known-hasheddata-01 : max$\int$ adset $\rightarrow$ max$\int$(adset ++ ahasheddata(a_HashedData));

end enrich

---

exagent =
enrich agentconsts **with**
predicates

exagent	:	agent;
is_terminal	:	agent;
is_smartcard	:	agent;

axioms

ex-DepositMachine : exagent(DepositMachine(n)) $\leftrightarrow$ n < NUMOFDEPOSITMACHINES;

ex-DepositMachine-simp : NUMOFDEPOSITMACHINES $\leq$ n $\rightarrow$ $\neg$ exagent(DepositMachine(n));

ex-CopyingMachine : exagent(CopyingMachine(n)) $\leftrightarrow$ n < NUMOFCOPYINGMACHINES;

ex-CopyingMachine-simp : NUMOFCOPYINGMACHINES $\leq$ n $\rightarrow$ $\neg$ exagent(CopyingMachine(n));

ex-Copycard : exagent(Copycard(n)) $\leftrightarrow$ n < NUMOFCOPYCARDS;

ex-Copycard-simp : NUMOFCOPYCARDS $\leq$ n $\rightarrow$ $\neg$ exagent(Copycard(n));

ex-user : exagent(user(n)) $\leftrightarrow$ n < NUMOFUSERS;

ex-attacker : exagent(attacker) $\leftrightarrow$ true;

is-terminal : is_terminal(ag) $\leftrightarrow$ is_DepositMachine(ag) $\vee$ is_CopyingMachine(ag);

is-smartcard : is_smartcard(ag) $\leftrightarrow$ is_Copycard(ag);

end enrich

---

messagelist =
actualize list-perm **with** messagefuns **by morphism**
elem $\rightarrow$ message; list $\rightarrow$ messagelist; a $\rightarrow$ msg; a_0 $\rightarrow$ msg_0; b $\rightarrow$ msg_1; c $\rightarrow$ msg_2; x $\rightarrow$ msgs; x_0 $\rightarrow$ $msgs_0$; y $\rightarrow$ $msgs_1$; z $\rightarrow$ $msgs_2$; y_0 $\rightarrow$ $msgs_3$; z_0 $\rightarrow$ $msgs_4$; x_1 $\rightarrow$ $msgs_5$; y_1 $\rightarrow$ $msgs_6$; z_1 $\rightarrow$ $msgs_7$; x_2 $\rightarrow$ $msgs_8$; y_2 $\rightarrow$ $msgs_9$; z_2 $\rightarrow$ $msgs_{10}$
end actualize

---

agentconsts =
enrich endpointPreds **with**

constants
NUMOFDEPOSITMACHINES : nat;
NUMOFCOPYINGMACHINES : nat;
NUMOFCOPYCARDS : nat;
NUMOFUSERS : nat;

axioms

NUMOFDEPOSITMACHINES-not-zero : NUMOFDEPOSITMACHINES $\neq$ 0;

NUMOFCOPYINGMACHINES-not-zero : NUMOFCOPYINGMACHINES $\neq$ 0;

NUMOFCOPYCARDS-not-zero : NUMOFCOPYCARDS $\neq$ 0;

NUMOFUSERS-not-zero : NUMOFUSERS $\neq$ 0;

end enrich

---

submessages =
enrich attackerdataset **with**
functions sub$\int$: attackerdata $\times$ attackerdataset $\rightarrow$ attackerdataset ;

axioms
sub-Load : sub$\int$(amessage(mkLoad(i, a_HashedData)), adset) =
{ahasheddata(a_HashedData)};

sub-RequestBalance : sub$\int$(amessage(mkRequestBalance), adset) = $\emptyset$;

sub-InvalidMessage : sub$\int$(amessage(mkInvalidMessage), adset) = $\emptyset$;

sub-Pay : sub$\int$(amessage(mkPay(i, a_Nonce)), adset) = {anonce(a_Nonce)};

sub-ResRequestBalance : sub$\int$(amessage(mkResRequestBalance(i)), adset) = $\emptyset$;

sub-ResPay :
sub$\int$(amessage(mkResPay(a_HashedData)), adset) = {ahasheddata(a_HashedData)};

sub-ResAuthenticate :
sub$\int$(amessage(mkResAuthenticate(a_Nonce)), adset) = {anonce(a_Nonce)};

sub-URequestBalance : sub$\int$(amessage(mkURequestBalance), adset) = $\emptyset$;

sub-URequestCopies : sub$\int$(amessage(mkURequestCopies(i)), adset) = $\emptyset$;

sub-UInsertMoney : sub$\int$(amessage(mkUInsertMoney(i)), adset) = $\emptyset$;

sub-Authenticate : sub$\int$(amessage(mkAuthenticate), adset) = $\emptyset$;

sub-UIssueCopies : sub$\int$(amessage(mkUIssueCopies(i)), adset) = $\emptyset$;

sub-UShowBalance : sub$\int$(amessage(mkUShowBalance(i)), adset) = $\emptyset$;

sub-ResLoad : sub$\int$(amessage(mkResLoad(i)), adset) = $\emptyset$;

sub-UResLoad : sub$\int$(amessage(mkUResLoad(i, i_0)), adset) = $\emptyset$;

sub-AuthData-data :
sub$\int$(adata(wrapAuthData2data(a_AuthData)), adset)
= {asecret(a_AuthData.passphrase)} $\cup$ {anonce(a_AuthData.challenge)};

sub-AuthData-HashData :
  $\text{sub}\int$(ahashdata(wrapAuthData2HashData(a_AuthData)), adset)
= {asecret(a_AuthData.passphrase)} ∪ {anonce(a_AuthData.challenge)};

sub-Secret : $\text{sub}\int$(asecret(a_Secret), adset) = ∅;

sub-Nonce : $\text{sub}\int$(anonce(a_Nonce), adset) = ∅;

sub-hashed : $\text{sub}\int$(ahasheddata(a_HashedData), adset) = ∅;

user-message-empty : is_user_message(msg) → $\text{sub}\int$(amessage(msg), adset) = ∅;

subint-nonce : $\text{sub}\int$(ad, adset ++ anonce(a_Nonce)) = $\text{sub}\int$(ad, adset);

subint-secret : $\text{sub}\int$(ad, adset ++ asecret(a_Secret)) = $\text{sub}\int$(ad, adset);

subint-hasheddataa : $\text{sub}\int$(ad, adset ++ ahasheddata(a_HashedData)) = $\text{sub}\int$(ad, adset);

end enrich

---

attackerdataset =
actualize set-union **with** attackerdata **by morphism**
  elem → attackerdata; set → attackerdataset; a → ad; b → ad_0; c → ad_1;
  s → adset; s_0 → adset_0; s_1 → adset_1; s_2 → adset_2
end actualize

---

endpointPreds =
enrich connections **with**
functions other-endpoint : endpoint × connection → endpoint ;
predicates
  endpoint-ok : endpoint;
  is-endpoint : endpoint × connection;
  is-endpoint : agent × connection;
  is-valid-port : agent × ports;
axioms
is-valid-port-def : is-valid-port(ag, port) ↔ endpoint-ok(ag ⊙ port);

other-endpoint-left : other-endpoint(endp_1, mk-connection(endp_1, endp_2)) = endp_2;

other-endpoint-right : other-endpoint(endp_2, mk-connection(endp_1, endp_2)) = endp_1;

is-endpoint-agent-def :
is-endpoint(ag, conn) ↔ conn.endpoint1.agent = ag ∨ conn.endpoint2.agent = ag;

is-endpoint-def :
is-endpoint(endp, conn) ↔ endp = conn.endpoint1 ∨ endp = conn.endpoint2;

endpoint-ok-def :
  endpoint-ok(endp)
↔ is_user(endp.agent) ∧ endp.port = CardOwner2DepositMachineDefaultPort
  ∨ is_user(endp.agent) ∧ endp.port = CardOwner2CopyingMachineDefaultPort
  ∨ is_Copycard(endp.agent) ∧ endp.port = Copycard2CopyingMachineDefaultPort
  ∨ is_Copycard(endp.agent) ∧ endp.port = Copycard2DepositMachineDefaultPort
  ∨ is_DepositMachine(endp.agent) ∧ endp.port = DepositMachine2CopycardDefaultPort

---

∨ is_DepositMachine(endp.agent) ∧ endp.port = DepositMachine2CardOwnerDefaultPort
∨ is_CopyingMachine(endp.agent) ∧ endp.port = CopyingMachine2CopycardDefaultPort
∨ is_CopyingMachine(endp.agent) ∧ endp.port = CopyingMachine2CardOwnerDefaultPort;

end enrich

---

attackerdata =
data specification
using messagefuns
attackerdata = amessage (. .message : message ;) **with** is_amessage
| asecret (. .secret : Secret ;) **with** is_asecret
| anonce (. .nonce : Nonce ;) **with** is_anonce
| adata (. .data : data ;) **with** is_adata
| ahashdata (. .hashdata : HashData ;) **with** is_ahashdata
| ahasheddata (. .hasheddata : HashedData ;) **with** is_ahasheddata
;
variables ad: attackerdata;
end data specification

---

connections =
actualize set-union **with** connection **by morphism**
elem → connection; set → connections; a → connection; b → connection_0;
c → connection_1; s → connectionset; s_0 → connectionset_0; s_1 → connectionset_1; s_2 → connectionset_2
end actualize

---

connection =
data specification
using endpoint
connection = mk-connection (. .endpoint1 : endpoint ; . .endpoint2 : endpoint ;);
variables conn, conn_0, conn_1, conn_2, conn_3: connection;
end data specification

---

messagefuns =
enrich message **with**
predicates

is_user_message	:	message;
is_valid_card_message	:	message;
. ∈ .	:	int × message;
. ∈ .	:	int × AuthData;

axioms is_valid_card_message-def :
is_valid_card_message(msg) ↔ ¬ is_user_message(msg) ∧ (∀ i. i ∈ msg → i ∈short);

int-in-Load : j ∈ mkLoad(i, a_HashedData) ↔ j = i;

int-in-RequestBalance : ¬ j ∈ mkRequestBalance;

int-in-Pay : j ∈ mkPay(i, a_Nonce) ↔ j = i;

int-in-ResRequestBalance : j ∈ mkResRequestBalance(i) ↔ j = i;

int-in-ResPay : ¬ j ∈ mkResPay(a_HashedData);

int-in-ResAuthenticate : ¬ j ∈ mkResAuthenticate(a_Nonce);

int-in-Authenticate : ¬ j ∈ mkAuthenticate;

int-in-ResLoad : j ∈ mkResLoad(i) ↔ j = i;

int-in-AuthData : j ∈ mkAuthData(i, a_Secret, a_Nonce, i_0) ↔ ¬ ¬ (j = i ∨ j = i_0);

is_user_message-def :
  is_user_message(msg)
↔ isURequestBalance(msg)
  ∨ isURequestCopies(msg)
  ∨ isUInsertMoney(msg)
  ∨ isUIssueCopies(msg)
  ∨ isUShowBalance(msg)
  ∨ isUResLoad(msg);

end enrich

---

endpoint =
data specification
using agentPreds, ports
endpoint = . ⊙ . **prio** 9 (. .agent : agent ; . .port : ports ;) **prio** 9;
variables endpoint, endp, $endp_0$, $endp_1$, $endp_2$, $endp_3$: endpoint;
end data specification

---

message =
data specification
using SecurityOperations
message = mkLoad (. .amount : int ; . .authterminal : HashedData ;) **with** isLoad
| mkRequestBalance **with** isRequestBalance
| mkPay (. .amount : int ; . .terminalchallenge : Nonce ;) **with** isPay
| mkResRequestBalance (. .balance : int ;) **with** isResRequestBalance
| mkResPay (. .authcard : HashedData ;) **with** isResPay
| mkResAuthenticate (. .cardletchallenge : Nonce ;) **with** isResAuthenticate
| mkURequestBalance **with** isURequestBalance
| mkURequestCopies (. .value : int ;) **with** isURequestCopies
| mkUInsertMoney (. .value : int ;) **with** isUInsertMoney
| mkAuthenticate **with** isAuthenticate
| mkUIssueCopies (. .value : int ;) **with** isUIssueCopies
| mkUShowBalance (. .value : int ;) **with** isUShowBalance
| mkResLoad (. .balance : int ;) **with** isResLoad
| mkUResLoad (. .balance : int ; . .amountToLoad : int ;) **with** isUResLoad
| mkInvalidMessage **with** isInvalidMessage

;
variables msg, msg_0, msg_1: message;
end data specification

SecurityOperations =
enrich data **with**
functions hash : HashData → HashedData ;
predicates verifyhash : HashedData × HashData;

axioms

hash-def : hash(a_HashData) = mkHashedData(a_HashData);

verifyhash-def :
verifyhash(a_HashedData, a_HashData) ↔ a_HashedData = mkHashedData(a_HashData);

end enrich

---

agentPreds =
enrich agent **with**
predicates is_statefulService : agent;

axioms

is_statefulService : ¬ is_statefulService(ag);

end enrich

---

ports =
data specification
ports = Copycard2CopyingMachineDefaultPort
| CopyingMachine2CopycardDefaultPort
| CopyingMachine2CardOwnerDefaultPort
| CardOwner2CopyingMachineDefaultPort
| Copycard2DepositMachineDefaultPort
| DepositMachine2CopycardDefaultPort
| DepositMachine2CardOwnerDefaultPort
| CardOwner2DepositMachineDefaultPort
;
variables port, $port_0$, $port_1$: ports;
end data specification

---

agent =
data specification
using int-pair

agent = DepositMachine (. .name : nat ;) **with** is_DepositMachine
| CopyingMachine (. .name : nat ;) **with** is_CopyingMachine
| Copycard (. .name : nat ;) **with** is_Copycard
| user (. .name : nat ;) **with** is_user
| attacker **with** is_attacker
;
variables ag, ag_0, ag_1: agent;
end data specification

---

data =
data specification
using bool, int-pair, string-data, byte, Secret, Nonce
AuthData = mkAuthData (. .instruction : int ;
. .passphrase : Secret ;
. .challenge : Nonce ;
. .amount : int ;
);
data = wrapAuthData2data (. .authData : AuthData ;) **with** isAuthData ;
HashData = wrapAuthData2HashData (. .authData : AuthData ;) **with** isAuthData ;
HashedData = mkHashedData (. .hash : HashData ;);
variables
a_data, a_data_0, a_data_1: data;
a_HashData, $a\_HashData_0$, $a\_HashData_1$: HashData;
a_AuthData, $a\_AuthData_0$, $a\_AuthData_1$: AuthData;
a_HashedData, $a\_HashedData_0$, $a\_HashedData_1$: HashedData;
end data specification

Nonce =
data specification
using string-data
Nonce = mkNonce (. .nonce : string ;);
variables a_Nonce, a_Nonce_0, a_Nonce_1: Nonce;
end data specification

Secret =
data specification
using string-data
Secret = mkSecret (. .secret : string ;);
variables a_Secret, a_Secret_0, a_Secret_1: Secret;
end data specification

Literaturverzeichnis

[1] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] Martín Abadi and Roger Needham. Prudent Engineering Practice for Cryptographic Protocols. *Journal of IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[3] Muhammad Alam, Ruth Breu, and Michael Breu. Model Driven Security for Web Services (MDS4WS). In *Proceedings of the 2004 8th International Multitopic Conference (INMIC)*, pages 498–505. IEEE Press, 2004.

[4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 1st edition, 2001.

[5] Ross J. Anderson and Roger M. Needham. Programming Satan's Computer. In *Computer Science Today*, pages 426–440. Springer, 1995.

[6] C. Apel, J. Repp, R. Rieke, and J. Steingruber. Modellbasiertes Testen der deutschen Gesundheitskarten. In *DACH Security 2007 - Bestandsaufnahme, Konzepte, Anwendungen, Perspektiven.*, pages 338–346, 2007.

[7] A. Armando, R. Carbone, L. Compagna, Keqin Li, and G. Pellegrino. Model-Checking Driven Security Testing of Web-Based Applications. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 361 –370. IEEE, 2010.

[8] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*. Springer, 2005.

[9] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal System Development with KIV. In *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[10] Michael Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction*. PhD thesis, University of Augsburg, Augsburg, Germany, 2005.

[11] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, pages 39–91, 2006.

[12] David A. Basin, Sebastian Mödersheim, and Luca Viganò. An On-the-Fly Model-Checker for Security Protocol Analysis. In *8th European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science, pages 253–270. Springer, 2003.

[13] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Lecture Notes in Computer Science. Springer-Verlag, 2007.

[14] Eurosmart Belgium. Eurosmart webseite. Website, 2012. `http://www.eurosmart.com`.

[15] Giampaolo Bella, Fabio Massacci, and Lawrence C. Paulson. Verifying the SET Purchase Protocols. *Journal on Automated Reasoning*, 36(1-2):5–37, 2006.

[16] Giampaolo Bella and Lawrence C. Paulson. Kerberos Version IV: Inductive Analysis of the Secrecy Goals. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *5th European Symposium on Research in Computer Security (ESORICS)*, LNCS 1485, pages 361–375. Springer, 1998.

[17] Giampaolo Bella and Lawrence C. Paulson. Accountability protocols: Formalized and verified. *ACM Transactions on Information and System Security (TISSEC)*, 9(2):138–161, 2006.

[18] Giampaolo Bella and Elvinia Riccobene. Formal Analysis of the Kerberos Authentication System. *Journal of Universal Computer Science*, 3:1337–1381, 1997.

[19] Giampaolo Bella and Elvinia Riccobene. A Realistic Environment for Crypto-Protocol Analyses by ASMs. In *Proceedings of the 5th International Workshop on Abstract State Machines*, pages 127–138, 1998.

[20] Bastian Best, Jan Jürjens, and Bashar Nuseibeh. Model-Based Security Engineering of Distributed Information Systems Using UMLsec. In *29th International Conference on Software Engineering (ICSE)*, pages 581–590. ACM, 2007.

[21] A.K. Bhattacharjee and R.K. Shyamasundar. Validated Code Generation for Activity Diagrams. In *Distributed Computing and Internet Technology*, volume 3816 of *Lecture Notes in Computer Science*, pages 508–521. Springer Berlin Heidelberg, 2005.

[22] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, pages 82–96. IEEE Computer Society, 2001.

[23] M. Borek, K. Stenzel, N. Moebius, and W. Reif. Model-Driven Development of Secure Service Applications introduced by a Banking System Example. Technical Report 2012-03, Universität Augsburg, 2012.

[24] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. North-Holland, Amsterdam, 1995.

[25] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[26] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[27] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104. ACM, 2007.

[28] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment.* Springer, 1st edition, 2003.

[29] Rolv Braek and Geir Melby. Model-Driven Service Engineering. In *Model Driven Software Development - Volume II of Research and Practice in Software Engineering*, pages 384–401. Springer-Verlag, 2005.

[30] Ruth Breu, Gerhard Popp, and Muhammad Alam. Model Based Development of Access Policies. *International Journal on Software Tools for Technology Transfer*, 9(5):457–470, 2007.

[31] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Security mutants for property-based testing. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs*, volume 6706 of *Lecture Notes in Computer Science*, pages 69–77. Springer, 2011.

[32] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[33] N. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *Formal Methods Europe (FME)*, Lecture Notes in Computer Science. Springer, 2003.

[34] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transaction on Computer Systems*, 8:18–36, 1990.

[35] Aisha Bushager and Mark Zwolinski. Modelling Smart Card Security Protocols in SystemC TLM. In *IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing*, pages 637–643. IEEE Computer Society, 2010.

[36] Jordi Cabot and Nicola Zannone. Towards an Integrated Framework for Model-driven Security Engineering. In *Proceedings of the Modeling Security Workshop (MoDELS).* CEUR Workshop Proceedings, 2008.

[37] Alessandro Coglio. An Approach to the Generation of High-Assurance Java Card Applets. In *Proc. 2nd Conference on High Confidence Software and Systems (HCSS'02)*, pages 69–77, March 2002.

[38] Alessandro Coglio. Code Generation for High-Assurance Java Card Applets. In *Proc. 3rd NSA Conference on High Confidence Software and Systems*, pages 85–93, 2003.

[39] Alessandro Coglio. Toward Automatic Generation of Provably Correct Java Card Applets. In *Proc. 5th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2003.

[40] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS)*, pages 108–128, 2004.

[41] Pierre Corbineau, Mathilde Duclos, and Yassine Lakhnech. Certified Security Proofs of Cryptographic Protocols in the Computational Model: An Application to Intrusion Resilience. In *First International Conference on Certified Programs and Proofs*, Lecture Notes in Computer Science, pages 378–393. Springer, 2011.

[42] Véronique Cortier and Steve Kremer, editors. *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*. IOS Press, 2011.

[43] Nicolas Courtois, Karsten Nohl, and Sean O'Neil. Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards. *IACR Cryptology ePrint Archive*, page 166, 2008.

[44] Cas J. F. Cremers, Pascal Lafourcade, and Philippe Nadeau. Comparing State Spaces in Automatic Protocol Analysis . In *Formal to Practical Security*, volume 5458/2009 of *Lecture Notes in Computer Science*, pages 70–94. Springer, 2009.

[45] C.J.F. Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *20th International Conference on Computer Aided Verification (CAV)*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[46] W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[47] Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[48] Martin Deubler, Johannes Grünbauer, Jan Jürjens, and Guido Wimmel. Sound development of secure service-based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 115–124. ACM, 2004.

[49] Premkumar T. Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, 22th International Conference on Software Engineering (ICSE), pages 227–239. ACM, 2000.

[50] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), 2008. Updated by RFCs 5746, 5878, 6176.

[51] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*, pages 350–357. IEEE Computer Society, 1981.

[52] Olivier Dubuisson. *ASN.1 - Communication Between Heterogeneous Systems*. Elsevier-Morgan Kaufmann, 2000.

[53] H. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. BI Wissenschaftsverlag, 1992.

[54] E.B. Fernandez, H. Washizaki, N. Yoshioka, and M. VanHilst. An Approach to Model-based Development of Secure and Reliable Systems. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES)*, pages 260 –265. IEEE Computer Society, 2011.

[55] Howard Foster, László Gönczy, Nora Koch, Philip Mayer, Carlo Montangero, and Dániel Varró. UML extensions for service-oriented systems. In *Rigorous software engineering for service-oriented systems*, pages 35–60. Springer-Verlag, 2011.

[56] Elizabeta Fourneret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jürjens, and Parvaneh Yousefi. Model-Based Security Verification and Testing for Smart-cards. In *Sixth International Conference on Availability, Reliability and Security (ARES)*, Lecture Notes in Computer Science, pages 272 –279. Springer, 2011.

[57] Martin Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series)*. Addison-Wesley Professional, October 2010.

[58] Bundesamt für Sicherheit in der Informationstechnik (BSI). Sicherheitsmechanismen in elektronischen Ausweisdokumenten. Website, 2012. `https://www.bsi.bund.de/ContentBSI/Themen/Elekausweise/Sicherheitsmechanismen/BAC/sicherheitsmechanismenBAC.html`.

[59] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994.

[60] Flavio D. Garcia, Gerhard Koning Gans, Ruben Muijrers, Peter Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MIFARE Classic. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 97–114, Berlin, Heidelberg, 2008. Springer-Verlag.

[61] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly Pickpocketing a Mifare Classic Card. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 3–15, Washington, DC, USA, 2009. IEEE Computer Society.

[62] Dominik Gessenharter and Martin Rauscher. Code Generation for UML 2 Activity Diagrams. In *Modelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 205–220. Springer Berlin Heidelberg, 2011.

[63] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[64] H. Grandy. *Formale Verifikation der Korrektheit sicherheitskritischer Java Anwendungen*. PhD thesis, University of Augsburg, Augsburg, Germany, 2008.

[65] H. Grandy, M. Bischof, G. Schellhorn, W. Reif, and K. Stenzel. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In *FM 2008: 15th Int. Symposium on Formal Methods*. Springer LNCS 5014, 2008.

[66] H. Grandy, K. Stenzel, and W. Reif. A Refinement Method for Java Programs. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Springer LNCS 4468, 2007.

[67] Holger Grandy, Robert Bertossi, Kurt Stenzel, and Wolfgang Reif. ASN1-light: A Verified Message Encoding for Security Protocols. In *Software Engineering and Formal Methods (SEFM)*, pages 195–204. IEEE Computer Society, 2007.

[68] Holger Grandy, Kurt Stenzel, and Wolfgang Reif. Refinement of Security Protocol Data Types to Java. In *PASSWORD Workshop 2006 at ECOOP 2006*, Nantes, France, 2006. URL: `http://research.ihost.com/password/`.

[69] Object Management Group. *Model Driven Architecture*, 2005. http://www.omg.org/mda/.

[70] Object Management Group. *Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF)*, 2010. http://www.omg.org/spec/ALF/.

[71] Object Management Group. *Query/View/Transformation Specification, Version 1.1*, 2011. http://www.omg.org/spec/QVT/1.1/.

[72] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (FUML)*, 2011. http://www.omg.org/spec/FUML/.

[73] Object Management Group. *Unified Modeling Language, Version 2.4.1*, 2011. http://www.omg.org/spec/UML/2.4.1/.

[74] Object Management Group. *Object Constraint Language, Version 2.3.1*, 2012. http://www.omg.org/spec/OCL/2.3.1/.

[75] Heise Medien Gruppe. Heise Security Online: PIN-Prüfung im EMV-Verfahren bei EC- und Kreditkarten ausgehebelt. Website, 2010. `http://www.heise.de/security/meldung/PIN-Pruefung-im-EMV-Verfahren-bei-EC-und-Kreditkarten-ausgehebelt-929528.html`.

[76] Heise Medien Gruppe. Heise Security Online. Website, 2012. `http://www.heise.de/security/`.

[77] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford Univ. Press, 1995.

[78] D. Haneberg. *Sicherheit von Smart Card-Anwendungen.* PhD thesis, University of Augsburg, Augsburg, Germany, 2006. ISBN: 3832515976 (in German).

[79] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Smart Card Applications: An ASM Approach. In *International Conference on integrated Formal Methods (iFM) 2007*, volume 4591 of *LNCS*. Springer, 2007.

[80] D. Haneberg, N. Moebius, W. Reif, G. Schellhorn, and K. Stenzel. Mondex: Engineering a Provable Secure Electronic Purse. *International Journal of Software and Informatics*, 5(1):159–184, 2011. `http://www.ijsi.org`.

[81] David Harel. Dynamic Logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[82] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[83] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer, 1986.

[84] Reiko Heckel and Marc Lohmann. Towards Model-Driven Testing. *Electronic Notes in Theoretical Computer Science*, 82(6):33–43, 2003.

[85] Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. WALDMEISTER - High-Performance Equational Deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.

[86] Gerard Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[87] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.

[88] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus – A Tool for Distributed Systems Specification. In *Proceedings of Formal Techniques in real-time and fault-tolerant Systems*, pages 467–470. Springer, 1996.

[89] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[90] *Application Programming Interface Java Card Platform, Version 2.2.1*, Oktober 2003. `www.oracle.com/technetwork/java/javame/javacard`.

[91] *Java Cryptography Architecture Reference Guide for Java Platform Standard Edition 6*, Oktober 2011. `http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[92] Jostein Jensen and Martin Gilje Jaatun. Security in Model Driven Development: A Survey. In *Sixth International Conference on Availability, Reliability and Security, ARES 2011*, Lecture Notes in Computer Science, pages 704–709. Springer, 2011.

[93] C. Jones and J. Woodcock, editors. *Formal Aspects of Computing*, volume 20 (1). Springer, January 2008.

[94] Audun Josang. Security Protocol Verification Using SPIN, 1995.

[95] J. Jürjens and R. Rumms. Model-based Security Analysis of the German Health Card Architecture. *Methods of Information in Medicine*, 47:409–416, 2008.

[96] Jan Jürjens. Security Analysis of Crypto-based Java Programs using Automated Theorem Provers. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 167–176. IEEE Computer Society, 2006.

[97] Jan Jürjens. Model-based Security Testing Using UMLsec: A Case Study. *Electronic Notes on Theor. Computer Science*, 220(1):93–104, 2008.

[98] Jan Jürjens, Loïc Marchal, Martín Ochoa, and Holger Schmidt. Incremental Security Verification for Evolving UMLsec models. In *In Proceedings of 7th European Conference on Modelling Foundations and Applications (ECMFA)*, Lecture Notes in Computer Science, pages 52–68. Springer, 2011.

[99] Jan Jürjens, Jörg Schreck, and Peter Bartmann. Model-based security analysis for mobile communications. In *30th International Conference on Software Engineering (ICSE)*, pages 683–692. ACM, 2008.

[100] Jan Jürjens, Jörg Schreck, and Yijun Yu. Automated Analysis of Permission-Based Security Using UMLsec. In *11th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 4961 of *Lecture Notes in Computer Science*, pages 292–295. Springer, 2008.

[101] Jan Jürjens and Guido Wimmel. Formally testing fail-safety of electronic purse protocols (extended abstract), 2001.

[102] Jan Jürjens and Guido Wimmel. Security Modelling for Electronic Commerce: The Common Electronic Purse Specifications. In *The First IFIP Conference on E-Commerce, E-Business, E-Government (I3E 2001)*, pages 489–506, 2001.

[103] Geylani Kardas and E. Turhan Tunali. Design and implementation of a smart card based healthcare information system. *Computer Methods and Programs in Biomedicine*, 81(1):66–78, 2006.

[104] K. Kasal, J. Heurix, and T. Neubauer. Model-Driven Development Meets Security: An Evaluation of Current Approaches. In *44th Hawaii International Conference on System Sciences (HICSS)*, pages 1–9. IEEE Computer Society, 2011.

[105] K. Katkalov. Modellgetriebenes Testen sicherheitskritischer Anwendungen. Diplomarbeit, Fakultät für Angewandte Informatik, Universität Augsburg, 2011.

[106] K. Katkalov, N. Moebius, K. Stenzel, M. Borek, and W. Reif. Model-Driven Testing of Security Protocols with SecureMDD. In *Fifth IFIP International Conference on New Technologies, Mobility and Security (NTMS 2012)*. IEEE XPlore, 2012.

[107] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006.

[108] Manuel Koch and Francesco Parisi-Presicce. Formal access control analysis in the software development process. In *Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering*, pages 67–76. ACM, 2003.

[109] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[110] Andreas Kraus, Alexander Knapp, and Nora Koch. Model-Driven Generation of Web Applications in UWE. In *In Proceedings of 3rd International Workshop on Model-Driven Web Engineering (MDWE)*, 2007.

[111] Christian Kroiss, Nora Koch, and Alexander Knapp. UWE4JSF: A Model-Driven Generation Approach for Web Applications. In *3rd Workhop on The Web and Requirements Engineering at ICWE 2012*, Lecture Notes in Computer Science, pages 493–496. Springer, 2009.

[112] M. Kuhlmann and M. Gogolla. Modeling and validating Mondex scenarios described in UML and OCL with USE. *Formal Aspects of Computing*, 20(1):79–100, 2008.

[113] RSA Laboratories. *PKCS v2.1: RSA Cryptography Standard*, 2002. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.

[114] Bruno Legeard and Mark Utting. *Practical Model-Based Testing*. Morgan Kaufmann, 2007.

[115] John Lloyd and Jan Jürjens. Security Analysis of a Biometric Authentication System Using UMLsec and JML. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 77–91. Springer-Verlag, 2009.

[116] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language, 5th International Conference*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2002.

[117] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using CSP And FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*, pages 147–166. Springer LNCS 1055, 1996.

[118] Check Point Software Technologies Ltd. Microsoft Server Message Block Vulnerability. Website, 2011. http://www.checkpoint.com/defense/advisories/public/announcement/1108-microsoft-smb.html.

[119] Paolo Maggi and Riccardo Sisto. Using SPIN to Verify Security Properties of Cryptographic Protocols. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 187–204. Springer-Verlag, 2002.

[120] Philip Mayer, Andreas Schroeder, and Nora Koch. MDD4SOA: Model-Driven Service Orchestration. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 203–212. IEEE Computer Society, 2008.

[121] Jay McCarthy and Shriram Krishnamurthi. Trusted Multiplexing of Cryptographic Protocols. In *Formal Aspects in Security and Trust*, volume 5983 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2010.

[122] J. McDermott. Visual security protocol modeling. In *Proceedings of the 2005 workshop on New security paradigms*, pages 97–109. ACM, 2005.

[123] Catherine Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[124] M. Memon, M. Hafner, and R. Breu. SECTISSIMO: A Platform-independent Framework for Security Services. In *Proceedings of the first International Modeling Security Workshop*. Springer LNCS, 2008.

[125] Microsoft. Microsoft Server Message Block Protocol. Website, 2012. http://msdn.microsoft.com/en-us/library/aa365233

[126] Microsoft. The security development lifecycle (sdl). Website, 2012. `http://www.microsoft.com/security/sdl/default.aspx`.

[127] SUN Microsystems. *Java Card Applet Developer's Guide. Java Card Version 2.1*, 1999.

[128] N. Moebius, M. Borek, K. Stenzel, and W.Reif. Secure MDD: Transformation of a UML application model to a formal specification. Technical Report 2012-10, Universität Augsburg, 2012. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[129] N. Moebius, M. Borek, K. Stenzel, and W.Reif. Secure MDD: Transformation of a UML application model to executable code. Technical Report 2012-11, Universität Augsburg, 2012. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[130] N. Moebius, D. Haneberg, G. Schellhorn, and W. Reif. A Modeling Framework for the Development of Provably Secure E-Commerce Applications. In *International Confe-*

rence on Software Engineering Advances (ICSEA) 2007. IEEE Press, 2007.

[131] N. Moebius, K. Stenzel, M. Borek, and W. Reif. Incremental Development of large, secure Smart Card Applications. In *Proceedings of the Workshop on Model-Driven Security at Models 2012*. ACM Digital Library, 2012.

[132] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. Model-Driven Code Generation for Secure Smart Card Applications. In *20th Australian Software Engineering Conference*. IEEE Press, 2009.

[133] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *Workshop on Secure Software Engineering, SecSE, at ARES 2009*. IEEE Press, 2009.

[134] N. Moebius, K. Stenzel, and W. Reif. Modeling Security-Critical Applications with UML in the SecureMDD Approach. *International Journal On Advances in Software*, 1(1), 2008.

[135] N. Moebius, K. Stenzel, and W. Reif. Generating Formal Specifications for Security-Critical Applications - A Model-Driven Approach. In *ICSE 2009 Workshop: International Workshop on Software Engineering for Secure Systems (SESS'09)*. IEEE/ACM Digital Libary, 2009.

[136] N. Moebius, K. Stenzel, and W. Reif. Formal Verification of Application-Specific Security Properties in a Model-Driven Approach. In *Proceedings of ESSoS 2010 - International Symposium on Engineering Secure Software and Systems*. Springer LNCS 5965, 2010.

[137] Lionel Montrieux, Jan Jürjens, Charles B. Haley, Yijun Yu, Pierre-Yves Schobbens, and Hubert Toussaint. Tool support for code generation from a UMLsec property. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 357–358, New York, NY, USA, 2010. ACM.

[138] Anderson Morais, Eliane Martins, Ana Cavalli, and Willy Jimenez. Security Protocol Testing Using Attack Trees. In *Proceedings of the International Conference on Computational Science and Engineering - Volume 02*, pages 690–697. IEEE Computer Society, 2009.

[139] Max Moser, Ortrun Ibens, Reinhold Letz, Joachim Steinbach, Christoph Goller, Johann Schumann, and Klaus Mayr. SETHEO and E-SETHEO - The CADE-13 Systems. *Journal of Automated Reasoning*, 18(2):237–246, 1997.

[140] Wojciech Mostowski. Systematic Development of JAVA CARD Applets, 2006.

[141] Steven J. Murdoch, Saar Drimer, Ross Anderson, and Mike Bond. Chip and PIN is Broken. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 433–446, 2010.

[142] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *Web Services Security: SOAP Message Security 1.0*. OASIS, 2004.

[143] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[144] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), 2005. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649.

[145] A. Nikseresht and K. Ziarati. MDA Based Framework for the Development of Smart Card Based Application. In *Proceedings of the International Multi Conference of Engineers and Computer Scientists*. Newswood Limited, 2011.

[146] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[147] U.S. Department Of Commerce/National Institute of Standards and Technology. *Federal Information Processing Standards: 46.3, The Data Encryption Standard (DES)*, 1999. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[148] U.S. Department Of Commerce/National Institute of Standards and Technology. *Federal Information Processing Standards: 180-2, Secure Hash Standard*, 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[149] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation, Version 1.1 Specification. Website, 2011. `http://www.omg.org/spec/QVT/`.

[150] Object Management Group (OMG). Webseite der Object Management Group. Website, 2012. `http://www.omg.org`.

[151] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[152] Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[153] Lawrence C. Paulson. Inductive Analysis of the Internet Protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.

[154] K. Peralta, A. Orozco, A. Zorzo, and F. Oliveira. Specifying Security Aspects in UML Models . In *First International Modeling Security Workshop*, Lecture Notes in Computer Science. Springer, 2009.

[155] Rolan Petrasch and Oliver Meimberg. *Model Driven Architecture*. dpunkt.verlag, 2006.

[156] D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 400 – 405, 2004.

[157] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-based tests for access control policies. In *1st International Conference on Software Testing, Verification, and Validation*, pages 338–347. IEEE Computer Society, 2008.

[158] Óscar Sánchez Ramón, Fernando Molina, Jesús García Molina, and José Ambrosio Toval Álvarez. ModelSec: A Generative Architecture for Model-Driven Security. *Journal of Universal Computer Science*, 15(15):2957–2980, 2009.

[159] Wolfgang Rankl and Wolfgang Effing. *Handbuch der Chipkarten*. Carl Hanser Verlag München Wien, 4. edition, 2002.

[160] W. Reif, G. Schellhorn, and K. Stenzel. Proving System Correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*. Townsville, Australia, Springer LNCS 1249, 1997.

[161] Wolfgang Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer-Verlag, Berlin, 1995.

[162] Julia Reznik, Tom Ritter, Rudolf Schreiner, and Ulrich Lang. Model Driven Development of Security Aspects. *Electronic Notes in Theoretical Computer Science*, pages 65–79, 2007.

[163] Dean Rosenzweig, Davor Runje, and Neva Slani. Privacy, Abstract Encryption and Protocols: An ASM Model - Part I. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Proceedings*, volume 2589 of *Lecture Notes in Computer Science*, pages 372–390. Springer, 2003.

[164] Peter Y. A. Ryan, Steve A. Schneider, Michael H. Goldsmith, Gavin Lowe, and Bill Roscoe. *The Modelling and analysis of security protocols: The CSP Approach*. Addison-Wesley, 2001.

[165] Vineet Saini, Qiang Duan, and Vamsi Paruchuri. Threat modeling using attack trees. *Journal of Computing Sciences in Colleges*, 23(4):124–131, 2008.

[166] G. Schellhorn. ASM refinement preserving invariants. In *Proceedings of the 14th International ASM Workshop, ASM'07*. Grimstad, Norway, 2007.

[167] G. Schellhorn. ASM Refinement Preserving Invariants. *J.UCS*, 14(12):1929–1948, 2008.

[168] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.

[169] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In U. Glässer J.-R. Abrial, editor, *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, pages 93 – 110. Springer LNCS 5115, 2009.

[170] Steve Schneider. Verifying authentication protocols with CSP. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 3–17. IEEE Computer Society, 1998.

[171] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. Wiley, 2nd edition, 1996.

[172] Bruce Schneier. Attack Trees. In *Dr. Dobb's Journal of Software Tools*, volume 24, pages 21–29, 1999.

[173] Andreas Schroeder and Philip Mayer. Verifying Interaction Protocol Compliance of Service Orchestrations. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 545–550. Springer-Verlag, 2008.

[174] Carsten Sensler, Michael Kunz, and Peter Schnell. Test automation with model-driven test script development. *OBJEKTspektrum*, 03 2006.

[175] Sandra Smith, Alain Beaulieu, and W. Greg Phillips. Modeling and verifying security protocols using UML 2. In *International Systems Conference (SysCon)*, pages 72 – 79. IEEE Computer Society, 2011.

[176] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis. *Journal of Computer Security*, 9(1-2):47–74, 2001.

[177] V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science*. Addison Wesley, 1991.

[178] Maria Spichkova, Florian Hölzl, and David Trachtenherz. Verified System Development with the AutoFocus Tool Chain. In *Proceedings 2nd Workshop on Formal Methods in the Development of Software (WS-FMDS)*, pages 17–24, 2012.

[179] Thomas Stahl and Markus Völter. *Model-Driven Software Development*. John Wiley and Sons, Ltd, 2006.

[180] K. Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik,URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, 2005.

[181] K. Stenzel, N. Moebius, and W. Reif. Formal verification of QVT transformations for code generation. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011*. Springer LNCS 6981, 2011.

[182] Bruno Tatibouet, Antoine Requet, Jean-Christophe Voisinet, and Ahmed Hammad. Java Card Code Generation from B Specifications. In *Formal Methods and Software Engineering*, volume 2885 of *Lecture Notes in Computer Science*, pages 306–318. Springer Berlin Heidelberg, 2003.

[183] Andreas Thums. *Formale Fehlerbaumanalyse*. PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German).

[184] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. SECFUZZ: Fuzz-testing Security Protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*, pages 7–13. IEEE Computer Society, 2012.

[185] Giesecke und Devrient GmbH. Homepage der Firma Giesecke und Devrient. Website, 2012. `http://www.gi-de.com/de/index.jsp`.

[186] M. Usman and A. Nadeem. Automatic Generation of Java Code from UML Diagrams using UJECTOR. In *International Journal of Software Engineering and its Applications*, 2009.

[187] D. Varró and A. Pataricza. Automated formal verification of model transformations. In *Critical Systems Development in UML,Proceedings of the UML'03 Workshop. Technical Report Universität München TUM-I0323*, pages 63–78, 2003.

[188] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. SPASS Version 2.0. In *18th International Conference on Automated Deduction*, pages 275–279, 2002.

[189] J. Woodcock. First Steps in the Verified Software Grand Challenge. *IEEE Computer Society*, 39(10):57–64, 2006.